Vedlegg A: Cluster

Innlesning

readbench.ld

```
## addtop

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "datastruct.h"

/* Datastructure to keep the transistors for each cell until we make
    instances in domains. */

typedef struct _terminal {
   char name[STRLEN];
   char placed;
   struct _terminal * next;
} Terminal;

typedef struct _cell {
   char name[STRLEN];
   Node * startnode;
   Edge * startedge;
   Terminal * startterminal;
   struct _cell * next;
} Cell;

Cell * startcell=NULL; /* Start of list of all cells. */

Cell * currentcell=NULL; /* The most recently read cell */
Node * currentnode=NULL; /* The most recently read transistor */
Edge * currentedge=NULL; /* The most recently read net */
Terminal * currentterminal=NULL; /* The most recently read terminal */
char auxname1[STRLEN],auxname2[STRLEN],auxname3[STRLEN],instance[STRLEN];
/* Auxilliary variables */

/* Some functions for operations on the cell datastructure */
```

```
Cell * InitializeCell(char * name){
   Cell * c = (Cell *)malloc(sizeof(Cell));
   if (!c){
      perror("malloc");
      exit(1);
   }
   strcpy(c→name,name);
   c→startnode=NULL;
   c→startedge=NULL;
   c→startterminal=NULL;
   c→next=NULL;
   return c;
}

void InsertCell(Cell * c){
   /* INVARIANT: c has NULL pointer for next.
        The cell is inserted at the start of the list */
   if (startcell){
      c→next=startcell;
      startcell=c;
   }
   else
      startcell=c;
}

Cell * FindCell(char * name, Cell * start){
   Cell * c=start;
   int found=0;
   while (c && !found){
      if (strcmp(c→name,name)==0) found=1;
      else c=c→next;
   }
   return c;
}

Terminal * InitializeTerminal(char * name){
   Terminal * t = (Terminal *)malloc(sizeof(Terminal));
   if (!t){
      perror("malloc");
      exit(1);
   }
   strcpy(t→name,name);
   t→placed='E'; /* E for empty */
   t→next=NULL;
   return t;
}
```

```
void InsertTerminal(Terminal * t, Terminal ** start){
   /* INVARIANT: t has NULL pointer for next.
        The terminal is inserted at the start of the list
        in a cell or a domain. */
   if (*start){
      t->next=*start;
      *start=t;
   }
   else
      *start=t;
}

void CopyTerminals(Terminal * old, Terminal ** new){
   Terminal * t1, *t2;
   *new=NULL;
   t1=old;
   while (t1){
      t2=InitializeTerminal(t1->name);
      InsertTerminal(t2,new);
      t1=t1->next;
   }
}

void AddNamesForTerminals(char * ext, Terminal * start){
   Terminal * t=start;
   while (t){
      strcat(t->name,"_");
      strcat(t->name,ext);
      t=t->next;
   }
}

void DeleteTerminals(Terminal * t){
   if(t)
      DeleteTerminals(t->next);
   else
      return;
   free(t);
}

void DeleteCells(Cell * c){
   if(c){
      DeleteNodes(c->startnode);
      DeleteEdges(c->startedge);
      DeleteTerminals(c->startterminal);
      DeleteCells(c->next);
   }
   else
```

```
      return;
   free(c);
}



/* Datastructure for domains. */

typedef struct _domain {
   char name[STRLEN];
   Node * startnode;
   Edge * startedge;
   Terminal * startterminal;
   Terminal * startinnerterminal;
   struct _domain * next;
} Domain;

Domain * startdomain=NULL; /* Start of list of all domains. */

Domain * currentdomain=NULL; /* The most recently read domain */
Domain * innerdomain=NULL; /* The most recently read inner domain */

Domain * workingdomain=NULL; /* Domain under construction */
Cell * workingcell=NULL; /* Cell under construction */
int domain; /* Indicates if we read a domain or a cell */
Terminal * auxterm1, * auxterm2; /* Auxilliaries */
/* These will be used when instantiating cells/inner domains. */


/* Some functions for operations on the domain datastructure */

Domain * InitializeDomain(char * name){
   Domain * d = (Domain *)malloc(sizeof(Domain));
   if (!d){
      perror("malloc");
      exit(1);
   }
   strcpy(d→name,name);
   d→startnode=NULL;
   d→startedge=NULL;
   d→startterminal=NULL;
   d→startinnerterminal=NULL;
   d→next=NULL;
   return d;
}

void InsertDomain(Domain * d){
   /* INVARIANT: d has NULL pointer for next.
```

```
        The domain is inserted at the start of the list */
    if (startdomain){
        d→next=startdomain;
        startdomain=d;
    }
    else
        startdomain=d;
}

Domain * FindDomain(char * name, Domain * start){
    Domain * d=start;
    int found=0;
    while (d && !found){
        if (strcmp(d→name,name)==0) found=1;
        else d=d→next;
    }
    return d;
}

void DeleteDomains(Domain * d){
    if(d){
        DeleteNodes(d→startnode);
        DeleteEdges(d→startedge);
        DeleteTerminals(d→startterminal);
        DeleteTerminals(d→startinnerterminal);
        DeleteDomains(d→next);
    }
    else
        return;
    free(d);
}


char * MakeName(char * name, char * blockname){
    if(strcmp(name,"vss")==0 || strcmp(name,"gnd")==0 )
        strcpy(name,"vgnd");
    else if (strcmp(name,"vdd")≠0 && strcmp(name,"vgnd")≠0){
        strcat(name,"_");
        strcat(name,blockname);
    }
    return name;
}



/* Here is the start of reading a cell */

## begin cell
```

```
InsertCell(currentcell=InitializeCell(zzs)); /* Initialize with the cell
                                                name we have just read */

## termlist cell name

/* This is the cells interface to the other cells/domains, so these names
   will possibly change. We store them in a list kept with each cell.
   The cellname is added to the name, to keep track of which cell its from. */
strcpy(auxname1,zzs);
MakeName(auxname1,currentcell→name);
InsertTerminal(InitializeTerminal(auxname1),&(currentcell→startterminal));

/* A terminal is also a signal, so we have to make one. The name of
   this signal may change completely later. */
InsertEdge(InitializeEdge(auxname1),&(currentcell→startedge));


## siglist name

/* A signal is only an edge. The name is signame_cellname. This may
   be extended as the cell is encapsulated further into (possibly
   several) layers of domains. Some of the benchmarks put the same
   name both in termlist and in siglist, so we have to check that the
   name does not exist in the edge list. */

strcpy(auxname1,zzs);
MakeName(auxname1,currentcell→name);

if(!FindEdge(auxname1,currentcell→startedge))
  InsertEdge(InitializeEdge(auxname1),&(currentcell→startedge));

## translist name

/* A transistor is a node with lots of attributes, but we will read
   this later. The name is treated similiar as for signals. */
strcpy(auxname1,zzs);
MakeName(auxname1,currentcell→name);
RInsertNode(currentnode=InitializeNode(auxname1,'E'),
            &(currentcell→startnode));


## translist source

strcpy(auxname1,zzs);
MakeName(auxname1,currentcell→name);
if ((currentedge=FindEdge(auxname1,currentcell→startedge))){
  currentnode→source=currentedge;
```

```
   MakePointerToNode(currentedge,currentnode);
}
else
printf("Reading translist from file: Node ");
printf("%s could not find net %s for
source.\n",currentnode→name,zzs);


## translist gate

strcpy(auxname1,zzs);
MakeName(auxname1,currentcell→name);
if ((currentedge=FindEdge(auxname1,currentcell→startedge))){
   currentnode→gate=currentedge;
   MakePointerToNode(currentedge,currentnode);
}
else
printf("Read from file: Node %s could not find net %s for
gate.\n",
        currentnode→name,zzs);


## translist drain

strcpy(auxname1,zzs);
MakeName(auxname1,currentcell→name);
if ((currentedge=FindEdge(auxname1,currentcell→startedge))){
   currentnode→drain=currentedge;
   MakePointerToNode(currentedge,currentnode);
}
else
printf("Read from file: Node %s could not find net %s for
drain.\n",
        currentnode→name,zzs);


## attribute translist integer

/* Integer attribute of a transistor has been read. */

if (strcmp(zzw,"width") == 0)
        currentnode→width=(double) zzn;
else if (strcmp(zzw,"length") == 0)
        currentnode→length=(double) zzn;

## attribute translist decimal

/* Decimal attribute of a transistor has been read. */
```

```
if (strcmp(zzw,"width") == 0)
        currentnode→width=zzn;
else if (strcmp(zzw,"length") == 0)
        currentnode→length=zzn;

## attribute translist string

/* String attribute has been read */

if (strcmp(zzw,"type") == 0)
        SetType(currentnode,zzs[0]);


## end cell

if (strcmp(zzs,currentcell→name) ≠ 0)
      printf("Inconsistency in data: cell begin %s ended with
cell end %s.\n",
              currentcell→name, zzs);

/* We have finished reading a cell */


/* Here we are about to start read a domain */

## begin domain

InsertDomain(currentdomain=InitializeDomain(zzs));
/* Initialize with the domain name we have just read */
workingdomain=NULL; /* These have to be NULL, because we do the */
workingcell=NULL; /* insertion "one round later". */


## iolist signal

/* This is the domains interface to the other domains, so these names
    will possibly change. We store them in a list kept with each domain.
    The domainname is added to the name, to keep track of which domain
    its from. */
strcpy(auxname1,zzs);
MakeName(auxname1,currentdomain→name);
InsertTerminal(currentterminal=InitializeTerminal(auxname1),
                    &(currentdomain→startterminal));

/* A terminal is also a signal, so we have to make one.
    The name of this signal may change completely later. */
InsertEdge(InitializeEdge(auxname1),&(currentdomain→startedge));
```

```
## iolist top
currentterminal→placed='T';


## iolist bottom
currentterminal→placed='B';


## row gate

/* We have read an instance of a cell or a domain. */
/* First we merge the structure we made from last gate in the row, with
    the rest of the structure in the domain. If it is the first time
    workingdomain and workingcell will be NULL pointers, so we do nothing.
    The test on domain is to ensure that we merge the correct structures. */
if (domain && workingdomain)
        MergeStructures(workingdomain→startnode,workingdomain→startedge,
                        &(currentdomain→startnode),&(currentdomain→startedge));
else if (workingcell)
        MergeStructures(workingcell→startnode,workingcell→startedge,
                        &(currentdomain→startnode),&(currentdomain→startedge));

if ((innerdomain=FindDomain(zzs1,startdomain))){
/* If it is a domain, we copy it, instantiate and get the iolist of the
    domain since these names are known to us now. */
   domain=1;
   strcpy(auxname1,zzs1);
   strcpy(instance,zzs2);
   MakeName(auxname1,zzs2);
   workingdomain=InitializeDomain(auxname1);
   CopyStructure(innerdomain→startnode,innerdomain→startedge,
                 &(workingdomain→startnode),&(workingdomain→startedge));
   CopyTerminals(innerdomain→startterminal,&(workingdomain→startterminal));
   AddNamesForNodes(zzs2,workingdomain→startnode);
   AddNamesForEdges(zzs2,workingdomain→startedge);
   AddNamesForTerminals(zzs2,workingdomain→startterminal);
   CopyTerminals(innerdomain→startterminal,&auxterm1);
/* We have to insert a whole list of inner terminals */
   if ((currentdomain→startinnerterminal && auxterm1)){
      auxterm2=auxterm1;
      while (auxterm2→next)
         auxterm2=auxterm2→next;
      auxterm2→next=currentdomain→startinnerterminal;
      currentdomain→startinnerterminal=auxterm1;
   }
   else
```

```
      currentdomain→startinnerterminal=auxterm1;
}
else if ((currentcell=FindCell(zzs1,startcell))){
   /* If it is a cell we do only the copying and instantiating part. */
   domain=0;
   strcpy(auxname1,zzs1);
   MakeName(auxname1,zzs2);
   workingcell=InitializeCell(auxname1);
   CopyStructure(currentcell→startnode,currentcell→startedge,
                    &(workingcell→startnode),&(workingcell→startedge));
   CopyTerminals(currentcell→startterminal,&(workingcell→startterminal));
   AddNamesForNodes(zzs2,workingcell→startnode);
   AddNamesForEdges(zzs2,workingcell→startedge);
   AddNamesForTerminals(zzs2,workingcell→startterminal);
   auxterm1=workingcell→startterminal;
}
else
printf("Domain %s is calling %s, which does not
exist.\n",currentdomain→name,zzs1);


## row signal

/* We have read one of the signals in this instance and we shall
    instantiate it by changing the corresponding name in the structure
    we have previously copied. */

if (domain){
   strcpy(auxname2,zzs);
   MakeName(auxname2,currentdomain→name);
   strcpy(auxname3,auxterm1→name);
   MakeName(auxname3,instance);
   if (ChangeEdgeNames(auxname3,auxname2,workingdomain→startedge))
      strcpy(auxterm1→name,auxname2);
   else
      printf("Failed to substitute %s for %s in domain instance %s
in %s.\n",
               auxname2, auxterm1→name, auxname1, currentdomain→name);
}
else{
   strcpy(auxname2,zzs);
   MakeName(auxname2,currentdomain→name);
   /* strcpy(auxname3,auxterm1->name);
        MakeName(auxname3,instance);*/
   if (ChangeEdgeNames(auxterm1→name,auxname2,workingcell→startedge))
      strcpy(auxterm1→name,auxname2);
   else
      printf("Failed to substitute %s for %s in cell instance %s
```

```
in %s.\n",
            auxname2, auxterm1→name, auxname1, currentdomain→name);
}
if(auxterm1)
      auxterm1=auxterm1→next;
else
printf("More actual than formal parameters in %s.\n",auxname1);


## row tail

/* We have to insert the last element as well. */
if (domain && workingdomain)
      MergeStructures(workingdomain→startnode,workingdomain→startedge,
                        &(currentdomain→startnode),&(currentdomain→startedge));
else if (workingcell)
MergeStructures(workingcell→startnode,workingcell→startedge,
                    &(currentdomain→startnode),&(currentdomain→startedge));


## end domain

if (strcmp(zzs,currentdomain→name) ≠ 0)
     printf("Inconsistency: domain begin %s ended with domain
end %s.\n",
            currentdomain→name, zzs);


## addend

void CleanUp(){
  Node * n;
  Terminal * t;
  Edge * e;
  /* All cell structures may be thrown away, ... */
  DeleteCells(startcell);

  /* and nearly all domain structures. */
  if (startdomain == currentdomain)
    DeleteDomains(startdomain→next);
  else
    printf("Internal error: No domains deleted due to changes in
program.\n");

  /* The global terminals must be converted to nodes */
  t=currentdomain→startterminal;
  while (t){
    RInsertNode(n=InitializeNode(t→name,'T'),&(currentdomain→startnode));
```

```
      e=FindEdge(t→name,currentdomain→startedge);
      n→source=e;
      MakePointerToNode(e, n);
      n→placed=t→placed;
      t=t→next;
   }

   /* We initialize the global variables, ... */
   startnode=currentdomain→startnode;
   startedge=currentdomain→startedge;

   /* and delete the last parts */
   DeleteTerminals(currentdomain→startterminal);
   DeleteTerminals(currentdomain→startinnerterminal);
   free(currentdomain);

   /* Edges connected to no nodes are feedthroughs that we didnt need, so we
        give a warning and delete them. */
   e=startedge;
   while (e){
      if (!e→size){
         printf("WARNING: Deleted feedthrough edge %s,connected to 0
nodes. \n",
                  e→name);
         RemoveEdge(e,&startedge);
      }
      e=e→next;
   }

   /* Some more global variables we should initialize. */
   numberofnodes=CountNodes(startnode);
   numberofedges=CountEdges(startedge);
}
```

Partisjonering

main.c

```
#include "datastruct.h"
#include "partition.h"
#include "cluster.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

extern void CleanUp();

extern int CutValue();
extern int MakeFirstPartition();
extern int KernighanLin();

#define NEIGH TRUE
#define NONEIGH FALSE

void MakeClusters(Node * startn, Edge * starte, Cluster ** startc)
{
    FindDirectlyConnectedNodes(startn, starte, startc, &DiffTest, NONEIGH);
    FindDirectlyConnectedNodes(startn, starte, startc, &PolyTest, NONEIGH);
    ConnectClusters(startn, starte, startc, NONEIGH);
    GrowClusters(startn, starte, startc);
    ConnectTerminals(startn, starte, startc);
    PrintClusterStatistics(*startc);
    printf("\nNumber of P-N twin pairs:  %d\n\n", numberoftwins);
}

void MakeClustersWithNeighbours(Node * startn, Edge * starte,
                                Cluster ** startc)
{
    FindPNTwins(startn, starte, startc);
    FindDirectlyConnectedNodes(startn, starte, startc, &DiffTest, NEIGH);
    FindDirectlyConnectedNodes(startn, starte, startc, &PolyTest, NEIGH);
    ConnectClusters(startn, starte, startc, NEIGH);
    GrowClusters(startn, starte, startc);
    ConnectTerminals(startn, starte, startc);
    PrintClusterStatistics(*startc);
    printf("\nNumber of P-N twin pairs:  %d\n\n", numberoftwins);
}

void ClearStructures(Node * startn, Cluster ** startc)
{
    DeleteClusters(*startc);
    *startc = NULL;
```

```
        while (startn) {
            startn→cluster = NULL;
            startn→partition = -1;
            startn→traversed = 0;
            startn→change = 0;
            startn = startn→next;
        }
}

void main()
{
        Cluster *startcluster = NULL;
        Node *newstartnode = NULL;
        Edge *newstartedge = NULL;
        int cut, i;
        srand((unsigned int) getpid()); /* Initialize rand() values */
        yyparse();
        CleanUp();
        PrintStatistics(startnode, startedge);
        MakeClustersWithNeighbours(startnode, startedge, &startcluster);
        ClearStructures(startnode, &startcluster);
        MakeClusters(startnode, startedge, &startcluster);
        ClearStructures(startnode, &startcluster);
        cut = MakeFirstPartition();
        printf("The initial cutvalue for two partitions is %d\n", cut);
        cut = KernighanLin();
        printf("The cutvalue after running Kernighan-Lin is %d\n\n",
cut);
}
```

datastruct.h

```
#ifndef DATASTRUCT_H
#define DATASTRUCT_H

#define TRUE 1
#define FALSE 0
#define STRLEN 60

/* Declarations of nodes and edges in this particular kind of graphs.
*/

/* A node has exactly three edges, because it is supposed to mimick a
    transistor.
*/
/* In addition we need to represent terminals for the edge of the chip
    as a special kind of node with only one edge. Note that the attributes
    are similar to the first attributes of Transistor, so we can use the same
    struct. Only the pointer _source_ will be used, since a terminal is only
    attached to one net.
*/

typedef struct _node {
      char name[STRLEN];
      char PNTtype;
      struct _cluster *cluster;
      int xpos, ypos;
      double width, length;
      int partition;
      char placed;
      int traversed, change;
      struct _edge *source, *gate, *drain;
      struct _node *next, *previous;
} Node;

/* An edge is connected to a not known number of nodes since we are dealing
    with hypergraphs. We represent this as a list of edgeelems.
*/

typedef struct _edgeelem {
      Node *nodeinlist;
      struct _edgeelem *nextelem;
} EdgeElem;

typedef struct _edge {
      char name[STRLEN];
      EdgeElem *firstelem;
```

```
    int size;
    struct _edge *next, *previous;
} Edge;

/* Actual definition af the global variables. The start points
   for the doubly linked lists of Nodes and Edges.
 */

Node *startnode;
Edge *startedge;
int numberofnodes, numberofedges;

/* Some useful funtions for initialization, insertion, updating and
   searching. */

extern Node *InitializeNode(char *name, char PNTtype);
extern void RInsertNode(Node * n, Node ** start);
extern Edge *InitializeEdge(char *name);
extern void InsertEdge(Edge * e, Edge ** start);
extern void MakePointerToNode(Edge * e, Node * n);
extern Node *GetEdgeNode(Edge * e, int num);
extern void SetType(Node * n, char t);
extern Node *FindNode(char *name, Node * start);
extern Edge *FindEdge(char *name, Edge * start);
extern int CountNodes(Node * start);
extern int CountEdges(Edge * start);
extern void PrintAllNodes(Node * start);
extern void PrintAllEdges(Edge * start);
extern void PrintStatistics(Node * startnode, Edge * startedge);
extern void CopyStructure(Node * oldnode, Edge * oldedge, Node ** newnode,
                          Edge ** newedge);
extern void MergeStructures(Node * sourcenode, Edge * sourceedge,
                            Node ** destnode, Edge ** destedge);
extern void AddNamesForNodes(char *ext, Node * start);
extern void AddNamesForEdges(char *ext, Edge * start);
extern int ChangeEdgeNames(char *old, char *new, Edge * start);
extern void DeleteNodes(Node * n);
extern void RemoveEdge(Edge * e, Edge ** start);
extern void DeleteEdges(Edge * e);
extern void DeleteStructure(Node * n, Edge * e);

/* We need several ways to remember which nodes we want to associate
   with which other nodes, so we make a general nodelist structure. */

typedef struct _nodelist {
    Node *nodepointer;
    struct _nodelist *next;
} NodeList;
```

```
extern NodeList *InitializeNodeList(Node * n);
extern void InsertNodeList(NodeList * nl, NodeList ** start);
extern NodeList *FindNodeList(Node * n, NodeList * start);
extern void DeleteNodeList(NodeList * nl);
#endif
```

datastruct.c

```
#include "datastruct.h"
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>

Node *InitializeNode(char *name, char PNTtype)
{
    Node *n = (Node *) malloc(sizeof(Node));
    if (!n) {
        perror("malloc");
        exit(1);
    }
    strcpy(n->name, name);
    n->PNTtype = PNTtype;
    n->cluster = NULL;
    n->xpos = -10000;
    n->ypos = -10000;
    n->width = 0;
    n->length = 0;
    n->partition = -1;
    n->placed = 'E';
    n->traversed = 0;
    n->change = 0;
    n->source = NULL;
    n->gate = NULL;
    n->drain = NULL;
    n->next = NULL;
    n->previous = NULL;
    return n;
}

void RInsertNode(Node * n, Node ** start)
{
    /* INVARIANT: n has NULL pointer for previous and next.
     The node is inserted at a random position in the list */
    Node *np;
    int i, num = 0, pos;
    if (!*start)
        *start = n;
    else {
        for (np = *start; np; np = np->next)
            num++;
        i = RAND_MAX / num;
        i *= num;
        while ((pos = rand()) >= i)
```

```
                continue;
            pos %= num;
            if (pos == 0) {
                n→next = *start;
                (*start)→previous = n;
                *start = n;
            } else {
                np = *start;
                for (i = 0; i < pos - 1; i++)
                    np = np→next;
                n→next = np→next;
                n→previous = np;
                np→next = n;
            }
        }
}

Edge *InitializeEdge(char *name)
{
    Edge *e = (Edge *) malloc(sizeof(Edge));
    if (!e) {
        perror("malloc");
        exit(1);
    }
    strcpy(e→name, name);
    e→firstelem = NULL;
    e→size = 0;
    e→next = NULL;
    e→previous = NULL;
    return e;
}

void InsertEdge(Edge * e, Edge ** start)
{
    /* INVARIANT: e has NULL pointer for previous and next.
      The edge is inserted at the start of the list */
    if (*start) {
        e→next = *start;
        (*start)→previous = e;
        *start = e;
    } else
        *start = e;
}

void MakePointerToNode(Edge * e, Node * n)
{
    /* insert a new edgeelem in the list
      INVARIANT: the last edgeelem in a list points to NULL */
```

```
    EdgeElem *ee = (EdgeElem *) malloc(sizeof(EdgeElem));
    if (!ee) {
        perror("malloc");
        exit(1);
    }
    ee->nodeinlist = n;
    ee->nextelem = e->firstelem;
    e->firstelem = ee;
    e->size += 1;
}

Node *GetEdgeNode(Edge * e, int num)
{
    /* num=0 will return the first element in the list. */
    EdgeElem *el = e->firstelem;
    while (num--)
        el = el->nextelem;
    return el->nodeinlist;
}

void SetType(Node * n, char t)
{
    n->PNTtype = (islower(t) ? t + 'A' - 'a' : t);
}

Node *FindNode(char *name, Node * start)
{
    Node *n = start;
    int found = 0;
    while (n && !found) {
        if (strcmp(n->name, name) == 0)
            found = 1;
        else
            n = n->next;
    }
    return n;
}

Edge *FindEdge(char *name, Edge * start)
{
    Edge *e = start;
    int found = 0;
    while (e && !found) {
        if (strcmp(e->name, name) == 0)
            found = 1;
        else
            e = e->next;
    }
```

```
    return e;
}

int CountNodes(Node * start)
{
    Node *n = start;
    int num = 0;
    while (n) {
        num++;
        n = n→next;
    }
    return num;
}

int CountEdges(Edge * start)
{
    Edge *e = start;
    int num = 0;
    while (e) {
        num++;
        e = e→next;
    }
    return num;
}

void PrintAllNodes(Node * start)
{
    Node *n = start;
    while (n) {
        printf("Node %s has type: %c.\nSize: width: %g length:
%g and position x=%d y=%d.\nIt belongs to partition %d.\n",
n→name, n→PNTtype, n→width, n→length, n→xpos, n→ypos, n→partition);
        if (n→source)
            printf("Source is %s.  ", n→source→name);
        if (n→gate)
            printf("Gate is %s.  ", n→gate→name);
        if (n→drain)
            printf("Drain is %s.", n→drain→name);
        printf("\n\n");
        n = n→next;
    }
}

void PrintAllEdges(Edge * start)
{
    Edge *e = start;
    EdgeElem *el;
    while (e) {
```

```
            printf("Edge %s is connected to these %d nodes:", e→name,
e→size);
            el = e→firstelem;
            while (el) {
                printf(" %s", el→nodeinlist→name);
                el = el→nextelem;
            }
            printf(".\n");
            e = e→next;
        }
}

void PrintStatistics(Node * startnode, Edge * startedge)
{
        Node *n = startnode;
        Edge *e;
        EdgeElem *el;
        int total = 0, num = 0, i, numT = 0, numP = 0, numN = 0;
        int hist[numberofnodes], onlyPnet[numberofnodes];
        int onlyNnet[numberofnodes], onlyTnet[numberofnodes];
        int PandTnet[numberofnodes], NandTnet[numberofnodes];
        int PandNnet[numberofnodes], PandNandTnet[numberofnodes];
        int histn = 0, onlyPn = 0, onlyNn = 0, onlyTn = 0, PandTn = 0, NandTn = 0,
          PandNn = 0;
        int PandNandTn = 0;
        for (i = 0; i < numberofnodes; i++)
            hist[i] = onlyPnet[i] = onlyNnet[i] = onlyTnet[i] = PandTnet[i] =
              NandTnet[i] = PandNnet[i] = PandNandTnet[i] = 0;
        for (e = startedge; e; e = e→next) {
            if (strcmp(e→name, "vdd") == 0)
                printf("vdd:  %d noder\n", e→size);
            if (strcmp(e→name, "vgnd") == 0)
                printf("vgnd:  %d noder\n", e→size);
            total += e→size;
            num++;
            hist[e→size]++;
            numP = numN = numT = 0;
            for (el = e→firstelem; el; el = el→nextelem) {
                if (el→nodeinlist→PNTtype == 'P' && !numP)
                    numP++;
                if (el→nodeinlist→PNTtype == 'N' && !numN)
                    numN++;
                if (el→nodeinlist→PNTtype == 'T' && !numT)
                    numT++;
            }
            if (numP)
                if (numN)
                    if (numT)
```

```
                    PandNandTnet[e→size]++;
                else
                    PandNnet[e→size]++;
            else if (numT)
                PandTnet[e→size]++;
            else
                onlyPnet[e→size]++;
        else if (numN)
            if (numT) {
                NandTnet[e→size]++;
            } else
                onlyNnet[e→size]++;
        else if (numT)
            onlyTnet[e→size]++;
        else
            printf("Error: Net with no nodes: %s\n", e→name);
    }
    if (num ≠ numberofedges)
        printf("Internal error: numberofedges=%d and actual
numbe is %d.\n",
            numberofedges, num);

    printf(" N E T - S T A T I S T I C S\n\n");
    printf(" Size onlyP onlyN onlyT PandT NandT PandN P,NandT
TOTAL\n");
    for (i = 0; i < numberofnodes; i++)
        if (hist[i]) {
            printf("%8d%8d%8d%8d%8d%8d%8d%8d%8d\n", i, onlyPnet[i],
                onlyNnet[i], onlyTnet[i], PandTnet[i], NandTnet[i],
                PandNnet[i], PandNandTnet[i], hist[i]);
            PandNandTn += PandNandTnet[i];
            PandNn += PandNnet[i];
            NandTn += NandTnet[i];
            PandTn += PandTnet[i];
            onlyTn += onlyTnet[i];
            onlyNn += onlyNnet[i];
            onlyPn += onlyPnet[i];
            histn += hist[i];
        }
    printf("-------------------------------------------------
--------\n");
    printf("SUMMARY:%8d%8d%8d%8d%8d%8d%8d%8d\n\n", onlyPn, onlyNn,
onlyTn,
        PandTn, NandTn, PandNn, PandNandTn, histn);

    printf("Average size of a net is %g.\n\n", (double) total / (double)
num);
    numP = numN = numT = 0;
```

```
    while (n) {
        if (n→PNTtype == 'T')
            numT++;
        else if (n→PNTtype == 'P')
            numP++;
        else if (n→PNTtype == 'N')
            numN++;
        else
            printf("Internal error: PrintStatistics: unknown node
type.\n");
        n = n→next;
    }
    if ((numT + numP + numN) ≠ numberofnodes)
        printf("Internal error: numberofnodes=%d and actual
number is %d.\n",
                numberofnodes, num);
    printf("%d terminals + %d P transistors + %d N transistor =
%d nodes.\n\n",
            numT, numP, numN, numberofnodes);
}

void CopyStructure(Node * oldnode, Edge * oldedge, Node ** newnode,
                   Edge ** newedge)
{
    Node *n1, *n2;
    Edge *e1;
    *newnode = NULL;
    *newedge = NULL;
    /* Copy the nodes first */
    n1 = oldnode;
    while (n1) {
        n2 = InitializeNode(n1→name, n1→PNTtype);
        n2→cluster = n1→cluster;
        n2→xpos = n1→xpos;
        n2→ypos = n1→ypos;
        n2→width = n1→width;
        n2→length = n1→length;
        n2→partition = n1→partition;
        n2→placed = n1→placed;
        n2→traversed = n1→traversed;
        n2→change = n1→change;
        RInsertNode(n2, newnode);
        n1 = n1→next;
    }
    /* Then copy the edges */
    e1 = oldedge;
    while (e1) {
        InsertEdge(InitializeEdge(e1→name), newedge);
```

```
        e1 = e1→next;
    }
    /* And last we copy the connections between nodes and edges */
    n2 = *newnode;
    while (n2) {
        if ((n1 = FindNode(n2→name, oldnode))) {
            if ((e1 = FindEdge(n1→source→name, *newedge))) {
                n2→source = e1;
                MakePointerToNode(e1, n2);
            } else
                printf("Node %s could not find net %s for
source.\n",
                        n1→name, n1→source→name);
            if ((e1 = FindEdge(n1→gate→name, *newedge))) {
                n2→gate = e1;
                MakePointerToNode(e1, n2);
            } else
                printf("Node %s could not find net %s for gate.\n",
                        n1→name, n1→gate→name);
            if ((e1 = FindEdge(n1→drain→name, *newedge))) {
                n2→drain = e1;
                MakePointerToNode(e1, n2);
            } else
                printf("Node %s could not find net %s for
drain.\n",
                        n1→name, n1→drain→name);
        } else
            printf("Internal error, can not find node %s in old
structure.\n",
                    n2→name);
        n2 = n2→next;
    }
}

void MergeStructures(Node * sourcenode, Edge * sourceedge,
                     Node ** destnode, Edge ** destedge)
{
    Node *n1 = sourcenode, *n2;
    Edge *e1, *e2;
    EdgeElem *el1;
    /* Puts all nodes into one big list. We merge nodes from sourcenode
      randomly into destnode. */
    if (!*destnode)
        *destnode = sourcenode;
    else
        while (n1) {
            n2 = n1;
            n1 = n1→next;
```

```
                n2→next = NULL;
                n2→previous = NULL;
                RInsertNode(n2, destnode);
            }

        /* Checks if an edge is present in both structures, and if it is removes
          one of them and resolves all pointers to and from nodes. Then one big
          list is created, and we throw away the old edge. */
        e1 = sourceedge;
        while (e1) {
            if ((e2 = FindEdge(e1→name, *destedge))) {
                el1 = e1→firstelem;
                while (el1) {
                    n1 = el1→nodeinlist;
                    MakePointerToNode(e2, n1);
                    if (n1→source == e1)
                        n1→source = e2;
                    else if (n1→gate == e1)
                        n1→gate = e2;
                    else if (n1→drain == e1)
                        n1→drain = e2;
                    else
                        printf("Internal error in data structure.\n");
                    el1 = el1→nextelem;
                }
                e2 = e1;
                e1 = e1→next;
                free(e2);
            } else {
                e2 = e1;
                e1 = e1→next;
                e2→next = NULL;
                e2→previous = NULL;
                InsertEdge(e2, destedge);
            }
        }
}

void AddNamesForNodes(char *ext, Node * start)
{
        Node *n = start;
        while (n) {
            strcat(n→name, "_");
            strcat(n→name, ext);
            n = n→next;
        }
}
```

```
void AddNamesForEdges(char *ext, Edge * start)
{
    /* NOTE: If the name is 'vdd' or 'vgnd' it is not changed, because
     these are special global names. These tests are done in MakeName
     when we do the reading, but we do them just to be sure */
    Edge *e = start;
    while (e) {
        if (strcmp(e->name, "vss") == 0 || strcmp(e->name, "gnd") == 0) {
            strcpy(e->name, "vgnd");
        } else if (strcmp(e->name, "vdd") ≠ 0 &&
                    strcmp(e->name, "vgnd") ≠ 0) {
            strcat(e->name, "_");
            strcat(e->name, ext);
        }
        e = e->next;
    }
}

int ChangeEdgeNames(char *old, char *new, Edge * start)
{
    Edge *e = start;
    int changed = 0;

    while (e && !changed) {
        if (strcmp(old, e->name) == 0) {
            strcpy(e->name, new);
            changed = 1;
        }
        e = e->next;
    }
    return changed;
}

void DeleteNodes(Node * n)
{
    if (n)
        DeleteNodes(n->next);
    else
        return;
    free(n);
}

void DeleteEdgeElems(EdgeElem * el)
{
    if (el)
        DeleteEdgeElems(el->nextelem);
    else
        return;
```

```
    free(el);
}

void RemoveEdge(Edge * e, Edge ** start)
{
    if (e == *start) {
        *start = e→next;
        (*start)→previous = NULL;
    } else {
        e→next→previous = e→previous;
        e→previous→next = e→next;
    }
    free(e);
}

void DeleteEdges(Edge * e)
{
    if (e) {
        DeleteEdgeElems(e→firstelem);
        DeleteEdges(e→next);
    } else
        return;
    free(e);
}

void DeleteStructure(Node * n, Edge * e)
{
    DeleteNodes(n);
    DeleteEdges(e);
}

/* Functions to operate on nodelists. */

NodeList *InitializeNodeList(Node * n)
{
    NodeList *nl = (NodeList *) malloc(sizeof(NodeList));
    if (!nl) {
        perror("malloc");
        exit(1);
    }
    nl→nodepointer = n;
    nl→next = NULL;
    return nl;
}

void InsertNodeList(NodeList * nl, NodeList ** start)
{
    if (*start) {
```

```
            nl→next = *start;
            *start = nl;
        } else
            *start = nl;
}

NodeList *FindNodeList(Node * n, NodeList * start)
{
        NodeList *nl = start;
        int found = 0;
        while (nl && !found) {
            if (nl→nodepointer == n)
                found = 1;
            else
                nl = nl→next;
        }
        return nl;
}

void DeleteNodeList(NodeList * nl)
{
        if (nl)
            DeleteNodeList(nl→next);
        else
            return;
        free(nl);
}
```

datastruct.h

```
#ifndef CLUSTER_H
#define CLUSTER_H

#include "datastruct.h"

/* Datastructure to keep a list of "clusters" – nodes that fit together. */

#define MAXCLUSTERSIZE 5

#define NCLUSTERS 32
#define PCLUSTERS 32

typedef struct _cluster {
    NodeList *nodes;
    int size;
    char type;
    struct _cluster *sameneigh, *otherneigh;
    int traversed;
    struct _cluster *next;
} Cluster;

extern Cluster *InitializeCluster();
extern void InsertCluster(Cluster * c, Cluster ** start);
extern Cluster *FindClusterForNode(Node * n, Cluster * start);
extern void MergeClusters(Cluster * c1, Cluster * c2, Cluster ** c);
extern int ExternalNets(Cluster * c);
extern void PrintAllClusters(Cluster * c);
extern void PrintClusterStatistics(Cluster * c);
extern void DeleteClusters(Cluster * c);
#endif
```

datastruct.c

```
#include "cluster.h"
#include "datastruct.h"
#include <stdlib.h>
#include <stdio.h>

/* Some functions for operations on a cluster structure. */

Cluster *InitializeCluster()
{
    Cluster *c = (Cluster *) malloc(sizeof(Cluster));
    if (!c) {
        perror("malloc");
        exit(1);
    }
    c->nodes = NULL;
    c->size = 0;
    c->type = 'E';
    c->sameneigh = NULL;
    c->otherneigh = NULL;
    c->traversed = FALSE;
    c->next = NULL;
    return c;
}

void InsertCluster(Cluster * c, Cluster ** start)
{
    if (*start) {
        c->next = *start;
        *start = c;
    } else
        *start = c;
}

Cluster *FindClusterForNode(Node * n, Cluster * start)
{
    Cluster *c = start;
    NodeList *nl;
    int found = 0;
    while (c && !found) {
        nl = c->nodes;
        while (nl && !found) {
            if (nl->nodepointer == n)
                found = 1;
            else
                nl = nl->next;
        }
```

```
            if (!found)
                c = c→next;
        }
        return c;
}

void MergeClusters(Cluster * c1, Cluster * c2, Cluster ** c)
{
        NodeList *nl;
        nl = c1→nodes;
        while (nl→next) {
            nl→nodepointer→cluster = c2;
            nl = nl→next;
        }
        nl→nodepointer→cluster = c2;
        nl→next = c2→nodes;
        c2→nodes = c1→nodes;
        c2→size += c1→size;
        if (c2→type == 'T')
            c2→type = c1→type;
        if ((*c) == c1)
            (*c) = c1→next;
        else {
            c2 = (*c);
            while (c2→next ≠ c1)
                c2 = c2→next;
            c2→next = c1→next;
        }
        free(c1);
}


int ExternalNets(Cluster * c)
{
        Edge *e[numberofedges];
        int edges[numberofedges];
        NodeList *nlt;
        Node *np;
        int nedges = 0, i, external = 0;
        for (i = 0; i < numberofedges; i++) {
            e[i] = NULL;
            edges[i] = 0;
        }
        for (nlt = c→nodes; nlt; nlt = nlt→next) {
            np = nlt→nodepointer;
            for (i = 0; i < nedges; i++)
                if (np→source == e[i])
                    break;
```

```
            if (i == nedges)
                e[nedges++] = np→source;
            edges[i]++;
            if (np→PNTtype ≠ 'T') {
                for (i = 0; i < nedges; i++)
                    if (np→gate == e[i])
                        break;
                if (i == nedges)
                    e[nedges++] = np→gate;
                edges[i]++;
                for (i = 0; i < nedges; i++)
                    if (np→drain == e[i])
                        break;
                if (i == nedges)
                    e[nedges++] = np→drain;
                edges[i]++;
            }
        }
        for (i = 0; i < nedges; i++)
            if (edges[i] < e[i]→size)
                external++;
        return external;
}


void PrintAllClusters(Cluster * c)
{
        Cluster *clt = c;
        NodeList *nlt;
        while (clt) {
            nlt = clt→nodes;
            printf("Cluster consists of ");
            while (nlt) {
                printf("%s ", nlt→nodepointer→name);
                nlt = nlt→next;
            }
            printf("\n");
            clt = clt→next;
        }
}

void PrintClusterStatistics(Cluster * c)
{
        Cluster *clt;
        NodeList *nlt;
        Node *n;
        int total = 0, num = 0, i = 0, numT = 0, numP = 0, numN = 0;
        int hist[numberofnodes], onlyPnet[numberofnodes], onlyNnet[numberofnodes];
```

```
int PandTnet[numberofnodes], NandTnet[numberofnodes];
int sameneighbourhist[3] =
{0, 0, 0};
int otherneighbourhist[3] =
{0, 0, 0};
int histn = 0, onlyPn = 0, onlyNn = 0, PandTn = 0, NandTn = 0;
Edge *e[numberofedges];
int edges[numberofedges], edgehist[numberofedges];
int nedges, numedges, totaledges = 0;

for (i = 0; i < numberofnodes; i++)
    hist[i] = onlyPnet[i] = onlyNnet[i] = PandTnet[i] = NandTnet[i] = 0;
for (i = 0; i < numberofedges; i++)
    edgehist[i] = 0;

for (clt = c; clt; clt = clt→next) {
    num = numT = numP = numN = numedges = 0;
    for (i = 0; i < numberofedges; i++) {
        e[i] = NULL;
        edges[i] = 0;
    }
    nedges = 0;

    for (nlt = clt→nodes; nlt; nlt = nlt→next) {
        num++;
        n = nlt→nodepointer;
        if (n→PNTtype == 'P' && !numP)
            numP++;
        if (n→PNTtype == 'N' && !numN)
            numN++;
        if (n→PNTtype == 'T' && !numT)
            numT++;
        for (i = 0; i < nedges; i++)
            if (n→source == e[i])
                break;
        if (i == nedges)
            e[nedges++] = n→source;
        edges[i]++;
        if (n→PNTtype ≠ 'T') {
            for (i = 0; i < nedges; i++)
                if (n→gate == e[i])
                    break;
            if (i == nedges)
                e[nedges++] = n→gate;
            edges[i]++;
            for (i = 0; i < nedges; i++)
                if (n→drain == e[i])
                    break;
```

```
                if (i == nedges)
                    e[nedges++] = n→drain;
                edges[i]++;
            }
        }
        if (numP) {
            if (clt→type ≠ 'P')
                printf("Error: Cluster with P nodes not of type
P.\n");
            if (numN)
                printf("Error: Cluster with both P and N
nodes.\n");
            else if (numT)
                PandTnet[num]++;
            else
                onlyPnet[num]++;
        } else if (numN) {
            if (clt→type ≠ 'N')
                printf("Error: Cluster with N nodes not of type
N.\n");
            if (numT)
                NandTnet[num]++;
            else
                onlyNnet[num]++;
        } else if (numT) {
            printf("Error: Cluster with only T nodes.\n");
            for (nlt = clt→nodes; nlt; nlt = nlt→next)
                printf("\t \t %s\n", nlt→nodepointer→source→name);
        } else
            printf("Error: Cluster with no nodes.\n");
        total++;
        hist[num]++;
        if (clt→sameneigh)
            sameneighbourhist[1]++;
        else
            sameneighbourhist[0]++;
        if (clt→otherneigh)
            otherneighbourhist[1]++;
        else
            otherneighbourhist[0]++;
        for (i = 0; i < nedges; i++)
            if (e[i]→size > edges[i])
                numedges++;
        edgehist[numedges]++;
        totaledges += numedges;
    }
    printf(" C L U S T E R - S T A T I S T I C S\n\n");
    printf(" Size onlyP onlyN PandT NandT TOTAL\n");
```

```
    for (i = 0; i < numberofnodes; i++)
        if (hist[i]) {
            printf("%8d%8d%8d%8d%8d%8d\n", i, onlyPnet[i], onlyNnet[i],
                    PandTnet[i], NandTnet[i], hist[i]);
            NandTn += NandTnet[i];
            PandTn += PandTnet[i];
            onlyNn += onlyNnet[i];
            onlyPn += onlyPnet[i];
            histn += hist[i];
        }
    printf("-------------------------------------------------------
--------\n");
    printf("SUMMARY:%8d%8d%8d%8d%8d\n\n", onlyPn, onlyNn, PandTn, NandTn,
histn);
    printf(" sameneighbours clusters\n");
    for (i = 0; i ≤ 2; i++)
        if (sameneighbourhist[i])
            printf("%12d%12d\n", i, sameneighbourhist[i]);
    printf("\n otherneighbours clusters\n");
    for (i = 0; i ≤ 2; i++)
        if (otherneighbourhist[i])
            printf("%12d%12d\n", i, otherneighbourhist[i]);
    printf("\n external nets clusters\n");
    for (i = 0; i < numberofedges; i++)
        if (edgehist[i])
            printf("%12d%12d\n", i, edgehist[i]);
    printf("\nTotal number of external nets: %d.\n", totaledges);
    printf("Average number of external net per cluster: %g\n",
            (double) totaledges / (double) histn);
}

void DeleteClusters(Cluster * c)
{
    if (c)
        DeleteClusters(c→next);
    else
        return;
    free(c);
}
```

partition.h

```
#ifndef PARTITION_H
#define PARTITION_H

#include "datastruct.h"
#include "cluster.h"

extern int numberoftwins;

/* Routines to do the neighbour clustering. */

extern int DiffTest(Node * n, Edge * e);
extern int PolyTest(Node * n, Edge * e);

extern void FindDirectlyConnectedNodes(Node * sn, Edge * se, Cluster ** retc,
                                       int (*test) (), int neigh);
extern void FindPNTwins(Node * n, Edge * e, Cluster ** retc);
extern void ConnectClusters(Node * n, Edge * e, Cluster ** retc, int neigh);
extern void GrowClusters(Node * n, Edge * e, Cluster ** retc);
extern void ConnectTerminals(Node * n, Edge * e, Cluster ** retc);
#endif
```

partition.c

```
#include "partition.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int numberoftwins = 0;

/* The first part of the partitioning is to find connected diffs and
   make clusters (or neighbours) of these nodes. Then we find
   connected polys and make clusters (or neighbours) of these
   nodes. We use the same subroutine for both tasks, buth supply the
   routine with different test routines for the different tasks*/

/* The test routine for diffs. */
int DiffTest(Node * n, Edge * e)
{
    return (n→source ≠ e && n→drain ≠ e);
}

/* The test routine for polys. */
int PolyTest(Node * n, Edge * e)
{
    return (n→gate ≠ e);
}

void FindDirectlyConnectedNodes(Node * sn, Edge * se, Cluster ** retc,
                                int (*conntest) (), int neigh)
{
    /* Returns a list of clusters where each cluster consists of
     transistors which can be very tight coupled, becauce these
     transistors are the only ones in a net and they are all in
     diff. The parameter neigh decides if we should make neighbour
     relations.*/
    Node *n[numberofnodes];
    Edge *e;
    Cluster *c = *retc, *clt[numberofnodes], *cl;
    int ni, ci, i, ok, found, val;
    char type = 'E';

    /* We check all the nets (edges). */
    for (e = se; e; e = e→next) {
        /* We are only interested in nets with more than two and less than
       MAXCLUSTERSIZE transistors. */
        if (e→size ≥ 2 && e→size ≤ MAXCLUSTERSIZE) {
            for (ni = 0; ni < numberofnodes; ni++) {
```

```
                n[ni] = NULL;
                clt[ni] = NULL;
            }
            ok = TRUE;
/* We find out if it is a P or N net (if it is neither we will see it
   later. */
            for (ni = 0; ni < e→size; ni++) {
                type = GetEdgeNode(e, ni)→PNTtype;
                if (type == 'P' || type == 'N')
                    break;
            }
            if (type ≠ 'P' && type ≠ 'N') {
                printf("Internal error: Net with no
transistors.\n");
                exit(1);
            }
            for (ni = 0; ni < e→size; ni++) {
                n[ni] = GetEdgeNode(e, ni);
/* The transistors have to be coupled correctly, so we call the
   supplied test routine. In addition we have to test that the type
   is correct.*/
                if ((*conntest) (n[ni], e) ||
                    (n[ni]→PNTtype ≠ type && n[ni]→PNTtype ≠ 'T')) {
                    ok = FALSE;
                    break;
                }
            }
/* If ok is TRUE, all the transistors are of the same type or of type
   terminal, and connected the way we wanted (in diff or in poly). */
            if (ok) {
/* We identify all the involved clusters. The same cluster may show
   up several times in the cluster array, but this does not matter. */
                ci = 0;
                for (ni = 0; ni < e→size; ni++)
                    if (n[ni]→cluster)
                        clt[ci++] = n[ni]→cluster;
                for (ni = 0; ni < e→size; ni++) {
/* We check if we find a cluster to insert into/merge with. */
                    val = (n[ni]→cluster ? n[ni]→cluster→size : 1);
                    found = FALSE;
                    for (ci = 0; clt[ci] ≠ NULL && !found; ci++)
                        if (clt[ci] ≠ n[ni]→cluster &&
                            clt[ci]→size + val ≤ MAXCLUSTERSIZE) {
                            found = TRUE;
                            ci--;
                        }
                    if (found)
                        if (!n[ni]→cluster) {
```

```
/* We have found a cluster to insert the single node into. */
                                    InsertNodeList(InitializeNodeList(n[ni]),
                                                   &clt[ci]→nodes);
                                    n[ni]→cluster = clt[ci];
                                    if (n[ni]→PNTtype ≠ 'T')
/* A terminal counts in an edge because the net depends on it, but it will
   not occupy much space in a layout, so we do not count it in a cluster */
                                        clt[ci]→size++;
                                } else {
/* We have found two clusters which are small enough to be merged. */
                                    for (i = 0; clt[i] ≠ n[ni]→cluster; i++);
                                    /* i is the index of the cluster to disappear. */
                                    MergeClusters(clt[i], clt[ci], &c);
                                    for (ci = 0; clt[ci]; ci++);
                                    clt[i] = clt[ci - 1];
                                    clt[ci - 1] = NULL;
                            } else if (!n[ni]→cluster) {
/* We did not find a cluster, so we have to make a new cluster with a
   single node. */
                                cl = InitializeCluster();
                                cl→nodes = InitializeNodeList(n[ni]);
                                cl→type = type;
                                if (n[ni]→PNTtype ≠ 'T')
                                    /* We do not count terminals in clusters. */
                                    cl→size = 1;
                                n[ni]→cluster = cl;
                                InsertCluster(cl, &c);
                                for (ci = 0; clt[ci]; ci++);
                                clt[ci] = cl;
                            } else
/* The node is already in a cluster, and we could not find a
   sufficiently small cluster to merge it with. If neigh is TRUE we
   set up some neighbour relation instead. */
                            if (neigh) {
                                for (ci = 0; clt[ci]; ci++)
                                    if (clt[ci] ≠ n[ni]→cluster &&
                                        clt[ci]→sameneigh == NULL &&
                                        n[ni]→cluster→sameneigh == NULL) {
                                        clt[ci]→sameneigh = n[ni]→cluster;
                                        n[ni]→cluster→sameneigh = clt[ci];
                                    }
                            }
                        }
                    }
                }
            }
            *retc = c;
}
```

```
void RememberTwins(Node * n1, Node * n2, Cluster ** c)
{
    /* Makes neighbours of the clusters the two nodes belong to. If one
     of them (or both) did not belong to a cluster, a new cluster is
     created for the node. This cluster will consist of this node only.
     Remember that we can not make one cluster of both nodes because
     they are of different types. */
    Cluster *c1, *c2;
    if ((!n1→cluster || (n1→cluster && n1→cluster→otherneigh == NULL)) &&
    (!n2→cluster || (n2→cluster && n2→cluster→otherneigh == NULL))) {
        if (n1→cluster)
            c1 = n1→cluster;
        else {
            c1 = InitializeCluster();
            c1→nodes = InitializeNodeList(n1);
            c1→type = n1→PNTtype;
            if (n1→PNTtype ≠ 'T')
                c1→size = 1;
            n1→cluster = c1;
            InsertCluster(c1, c);
        }
        if (n2→cluster)
            c2 = n2→cluster;
        else {
            c2 = InitializeCluster();
            c2→nodes = InitializeNodeList(n2);
            c2→type = n2→PNTtype;
            if (n2→PNTtype ≠ 'T')
                c2→size = 1;
            n2→cluster = c2;
            InsertCluster(c2, c);
        }
        c1→otherneigh = c2;
        c2→otherneigh = c1;
        numberoftwins++;
    } else
        printf("Could not find suitable cluster to rememeber.
\n");
}

void FindPNTwins(Node * n, Edge * e, Cluster ** retc)
{
    Edge *follow;
    EdgeElem *el;
    Node *n1;
    while (n) {
```

```
            follow = NULL;
            if (n→PNTtype ≠ 'P') {
                n = n→next;
                continue;
            }
            if (strcmp(n→source→name, "vdd") == 0
                || strcmp(n→source→name, "vgnd") == 0)
                follow = n→drain;
            if (strcmp(n→drain→name, "vdd") == 0
                || strcmp(n→drain→name, "vgnd") == 0)
                follow = n→source;
            if (follow && follow→size > 2) {
                el = follow→firstelem;
                while (el) {
                    n1 = el→nodeinlist;
                    if (n1→PNTtype == 'N' &&
                        (n1→source == follow || n1→drain == follow))
                        if (strcmp(n1→source→name, "vdd") == 0 ||
                            strcmp(n1→source→name, "vgnd") == 0 ||
                            strcmp(n1→drain→name, "vdd") == 0 ||
                            strcmp(n1→drain→name, "vgnd") == 0)
                            RememberTwins(n, n1, retc);
                    el = el→nextelem;
                }
            }
            n = n→next;
        }
}

int ConnectionFormulae(Cluster * c1, Cluster * c2, int neigh)
{
    NodeList *nlt;
    Edge *e[numberofedges];
    int i, num = 0, commonedges = 0, neighbouredges = 0;
    for (nlt = c1→nodes; nlt; nlt = nlt→next) {
        for (i = 0; i < num; i++)
            if (nlt→nodepointer→source == e[i])
                break;
        if (i == num)
            e[num++] = nlt→nodepointer→source;
        if (nlt→nodepointer→PNTtype ≠ 'T') {
            for (i = 0; i < num; i++)
                if (nlt→nodepointer→gate == e[i])
                    break;
            if (i == num)
                e[num++] = nlt→nodepointer→gate;
            for (i = 0; i < num; i++)
                if (nlt→nodepointer→drain == e[i])
```

```
                    break;
            if (i == num)
                e[num++] = nlt→nodepointer→drain;
        }
    }
    if (neigh) {
        if (c1→sameneigh)
            for (nlt = c1→sameneigh→nodes; nlt; nlt = nlt→next) {
                for (i = 0; i < num; i++)
                    if (nlt→nodepointer→source == e[i])
                        break;
                if (i == num)
                    e[num++] = nlt→nodepointer→source;
                if (nlt→nodepointer→PNTtype ≠ 'T') {
                    for (i = 0; i < num; i++)
                        if (nlt→nodepointer→gate == e[i])
                            break;
                    if (i == num)
                        e[num++] = nlt→nodepointer→gate;
                    for (i = 0; i < num; i++)
                        if (nlt→nodepointer→drain == e[i])
                            break;
                    if (i == num)
                        e[num++] = nlt→nodepointer→drain;
                }
            }
        if (c1→otherneigh)
            for (nlt = c1→otherneigh→nodes; nlt; nlt = nlt→next) {
                for (i = 0; i < num; i++)
                    if (nlt→nodepointer→source == e[i])
                        break;
                if (i == num)
                    e[num++] = nlt→nodepointer→source;
                if (nlt→nodepointer→PNTtype ≠ 'T') {
                    for (i = 0; i < num; i++)
                        if (nlt→nodepointer→gate == e[i])
                            break;
                    if (i == num)
                        e[num++] = nlt→nodepointer→gate;
                    for (i = 0; i < num; i++)
                        if (nlt→nodepointer→drain == e[i])
                            break;
                    if (i == num)
                        e[num++] = nlt→nodepointer→drain;
                }
            }
    }
    for (nlt = c2→nodes; nlt; nlt = nlt→next) {
```

```
            for (i = 0; i < num; i++)
                if (nlt→nodepointer→source == e[i]) {
                    commonedges++;
                    break;
                }
            if (nlt→nodepointer→PNTtype ≠ 'T') {
                for (i = 0; i < num; i++)
                    if (nlt→nodepointer→gate == e[i]) {
                        commonedges++;
                        break;
                    }
                for (i = 0; i < num; i++)
                    if (nlt→nodepointer→drain == e[i]) {
                        commonedges++;
                        break;
                    }
            }
        }
        if (neigh) {
            if (c1→sameneigh && c1→sameneigh == c2→sameneigh)
                neighbouredges++;
            if (c1→otherneigh && c1→otherneigh == c2→otherneigh)
                neighbouredges++;
            return (2 * commonedges + neighbouredges);
        } else
            return (2 * commonedges);
}

void ConnectClustersOfType(Node * n, Edge * e, Cluster ** retc, char type,
                                    int maxclusters, int size, int neigh)
{
        /* Small clusters, from size nodes up to MAXCLUSTERSIZE-size; are
         joined. We are guaranteed to have few enough clusters >= size
         after running this procedure, but in addition we may have som
         clusters with fewer nodes. This will be handled later, in a
         subsequent call to this procedure. */

        Cluster *connectcluster[numberofnodes];
        int connectvalue[numberofnodes][numberofnodes];
        int i, j, numberofclusters = 100000, numberofterminals = 0;
        int bigclusters = 0, maxvalue, merge;
        Cluster *c, *best1, *best2;
        NodeList *nlt;
        Node *np;

        while (numberofclusters + bigclusters - numberofterminals
                > maxclusters + 1) {
            best1 = best2 = NULL;
```

```
maxvalue = -10000;
bigclusters = numberofterminals = 0;
for (i = 0; i < numberofnodes; i++) {
        connectcluster[i] = NULL;
        for (j = 0; j < numberofnodes; j++)
                connectvalue[i][j] = 0;
}
i = 0;
for (c = *retc; c; c = c→next)
        if (c→type == type && c→size > MAXCLUSTERSIZE - size)
                bigclusters++;
        else if (c→type == type &&
                        c→size ≥ size && c→size ≤ MAXCLUSTERSIZE - size)
                connectcluster[i++] = c;
        else if (c→type == 'T' &&
                        c→size ≥ size && c→size ≤ MAXCLUSTERSIZE - size) {
                connectcluster[i++] = c;
                numberofterminals++;
        }
numberofclusters = i;
for (i = 0; i < numberofclusters; i++)
        for (j = 0; j < i; j++) {
                connectvalue[i][j] =
                        ConnectionFormulae(connectcluster[i], connectcluster[j],
                                                neigh);
                if (connectcluster[i]→type ≠ 'T' ||
                        connectcluster[j]→type ≠ 'T')
                   if (maxvalue < connectvalue[i][j] &&
                        connectvalue[i][j] ≥ 0)
                      if (connectcluster[i]→size + connectcluster[j]→size
                                ≤ MAXCLUSTERSIZE) {
                                maxvalue = connectvalue[i][j];
                                best1 = connectcluster[i];
                                best2 = connectcluster[j];
                                merge = TRUE;
                      } else if (neigh && !connectcluster[i]→sameneigh &&
                                        !connectcluster[j]→sameneigh) {
                                maxvalue = connectvalue[i][j];
                                best1 = connectcluster[i];
                                best2 = connectcluster[j];
                                merge = FALSE;
                      }
        }

if (numberofclusters + bigclusters > maxclusters)
        if (best1)
                if (merge)
                        MergeClusters(best1, best2, retc);
```

```
                    else {
                        best1→sameneigh = best2;
                        best2→sameneigh = best1;
                } else if (numberofclusters + bigclusters - numberofterminals
                            > maxclusters) {
                    printf("Not possible to join enough clusters.");
                    printf("Adjust parameters?\n");
                    exit(1);
                }
        }
}

void ConnectClusters(Node * n, Edge * e, Cluster ** retc, int neigh)
{
        /* We have to connect both the P clusters and the N clusters of size 2
          up to MAXCLUSTERSIZE-2. */
        ConnectClustersOfType(n, e, retc, 'P', PCLUSTERS, 2, neigh);
        ConnectClustersOfType(n, e, retc, 'N', NCLUSTERS, 2, neigh);
}

void MakeSingletonClusters(Node * n, Cluster ** retc)
{
        Cluster *c;
        while (n) {
                if (!n→cluster) {
                        c = InitializeCluster();
                        c→nodes = InitializeNodeList(n);
                        c→type = n→PNTtype;
                        if (n→PNTtype ≠ 'T')
                                c→size = 1;
                        n→cluster = c;
                        InsertCluster(c, retc);
                }
                n = n→next;
        }
}

void GrowClusters(Node * n, Edge * e, Cluster ** retc)
{
/* First we make singleton clusters of single nodes. */
        MakeSingletonClusters(n, retc);
/* We have to connect both the P clusters and the N clusters. */
        ConnectClustersOfType(n, e, retc, 'P', PCLUSTERS, 1, FALSE);
        ConnectClustersOfType(n, e, retc, 'N', NCLUSTERS, 1, FALSE);
}

void ConnectTerminals(Node * n, Edge * e, Cluster ** retc)
{
```

```
/* We have some terminals that are alone in clusters. These clusters
   will have 0 size. We locate them and merge them with the cluster
   which they have most common edges with. */
    Cluster *c, *best;
    EdgeElem *el;
    int maxval, value;
    for (c = *retc; c; c = c->next)
        if (c->size == 0) {
            maxval = -10000;
            best = NULL;
            for (el = c->nodes->nodepointer->source->firstelem; el;
                 el = el->nextelem)
                if (c != el->nodeinlist->cluster) {
                    value = ConnectionFormulae(c, el->nodeinlist->cluster,
                                               FALSE);
                    if (maxval < value && value >= 0) {
                        maxval = value;
                        best = el->nodeinlist->cluster;
                    }
                }
            if (best) {
                MergeClusters(c, best, retc);
            } else
              printf("Can not find cluster to merge terminal
cluster with.\n");
        }
}
```

"Kernighan og Lin"-algoritmen

kl.c

```
#include "datastruct.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

/* kl - The Kernighan & Lin algorithm for graph partitioning. */
/* Works directly on the global structure. */

int globalcut;

int MakeFirstPartition()
{
    /* Just split the global structure into two partitions of equal size. */
    Node *n = startnode;
    int i;
    for (i = 0; i < numberofnodes / 2; i++) {
        n->partition = 0;
        n = n->next;
    }
    while (i++ < numberofnodes) {
        n->partition = 1;
        n = n->next;
    }
    return globalcut = CutValue();
}

int CutValue()
{
    int part, crossing = 0;
    Edge *e = startedge;
    EdgeElem *el;
    while (e) {
        el = e->firstelem;
        part = el->nodeinlist->partition;
        el = el->nextelem;
        while (el) {
            if (part != el->nodeinlist->partition) {
                crossing++;
                break;
            }
            el = el->nextelem;
        }
        e = e->next;
    }
```

```
    return crossing;
}

int GainValue(Node * n, EdgeElem * el, Node * othern)
{
    int same = FALSE, other = FALSE, othernode = FALSE;

    while (el) {
        if (el→nodeinlist→partition == n→partition && el→nodeinlist ≠ n)
            same = TRUE;
        if (el→nodeinlist→partition ≠ n→partition)
            other = TRUE;
        if (el→nodeinlist == othern)
            othernode = TRUE;
        el = el→nextelem;
    }
    if (!other)
        return -1;
    else if (!same && !othernode)
        return 1;
    else
        return 0;
}


int SwapBestPair(Node ** ret0, Node ** ret1)
{
    /* Swaps a pair in the global structure and marks the nodes as swapped.
     Returns 0 if round is over, eg. if all nodes already are swapped.
     Returns pointers to the nodes that are swapped, if any. */
    Node *n0, *n1;
    Node *best0 = NULL, *best1 = NULL;
    int maxgain = -10000, gain;

    for (n0 = startnode; n0; n0 = n0→next)
        if (n0→partition == 0 && !n0→traversed)
            for (n1 = startnode; n1; n1 = n1→next)
                if (n1→partition == 1 && !n1→traversed) {
                    gain = 0;
                    if (n0→source)
                        gain += GainValue(n0, n0→source→firstelem, n1);
                    if (n0→gate)
                        gain += GainValue(n0, n0→gate→firstelem, n1);
                    if (n0→drain)
                        gain += GainValue(n0, n0→drain→firstelem, n1);
                    if (n1→source)
                        gain += GainValue(n1, n1→source→firstelem, n0);
                    if (n1→gate)
```

```
                        gain += GainValue(n1, n1→gate→firstelem, n0);
                    if (n1→drain)
                        gain += GainValue(n1, n1→drain→firstelem, n0);
                    if (maxgain < gain) {
                        maxgain = gain;
                        best0 = n0;
                        best1 = n1;
                    }
                }
    if (best0) {
        best0→partition = 1;
        best1→partition = 0;
        best0→traversed = TRUE;
        best1→traversed = TRUE;
        globalcut = globalcut - maxgain;
        *ret0 = best0;
        *ret1 = best1;
        return 1;
    } else
        return 0;
}

int KernighanLin()
{
    Node *n0, *n1;
    NodeList *rollback = NULL; /* Each swap generates two elements
                                  in this list. */
    int cut, oldbestcut, bestcut;
    globalcut = bestcut = CutValue();
    /* Starting a round. */
    do {
        oldbestcut = bestcut;
        while (SwapBestPair(&n0, &n1)) {
            cut = globalcut;

            if (cut < bestcut) {
                bestcut = cut;
                DeleteNodeList(rollback);
                rollback = NULL;
            } else {
                InsertNodeList(InitializeNodeList(n0), &rollback);
                InsertNodeList(InitializeNodeList(n1), &rollback);
            }
        }

        while (rollback) {
            n0 = rollback→nodepointer;
            if (n0→partition == 0)
```

```
                n0→partition = 1;
            else
                n0→partition = 0;
            rollback = rollback→next;
        }
        DeleteNodeList(rollback);
        rollback = NULL;
        globalcut = bestcut;
        for (n0 = startnode; n0; n0 = n0→next)
            n0→traversed = FALSE;
    } while (bestcut < oldbestcut);
    return globalcut;
}
```

Vedlegg B: Cell

main.h

```
/* -*- C++ -*- $Id: main.h,v 1.12 1991/10/11 14:58:47 dash Exp $ */

#ifndef _main_h
#define _main_h

// Here follows defines to customize program


// Define PROFILING to profile the program, use 'gprof ./ivc mon.out'

// to get a flat profile of where the time in the program is used

// #define PROFILING


// Define IV to get InterViews-interface unless NO_IV or PROFILING is defined

#ifndef NO_IV
#ifndef PROFILING
#define IV
#endif /* PROFILING */
#endif /* NO_IV */

#include <stream.h>
#include <stdlib.h>

// extern "C" { int system(const char*); }


#ifndef NULL
#define NULL 0 // Common constants...

#endif
#define FALSE 0
#define TRUE 1
```

```
#define min(x,y) (x<y ? x : y)
#define max(x,y) (x>y ? x : y)

typedef int Bool; // Use Bool to signal that its a Boolean value


void wrapper(); // This is here for easy debuging


// Explanation of DEBUG_LVL values:

// 1 - Only big modules, 2 - Some details, 3 - Lots of detail, 4 - Everything

#ifndef DEBUG_LVL
#define DEBUG_LVL 100
#endif

// Usage: DEBOUT(111, DV2(x,y));

#define DEBOUT(num,dtxt)\
  if (DEBUG_LVL ≤ num) \
      cerr ≪ "debug[" ≪ __FILE__ ≪ ":" ≪ __LINE__ ≪ "]  " ≪ dtxt ≪ "\n"

#define DEBVAR(num,dtxt)\
  if (DEBUG_LVL ≤ num) \
  cerr ≪ "debug[" ≪ __FILE__ ≪ ":" ≪ __LINE__ ≪ "]  " \
  ≪ #dtxt ≪ "  =  " ≪ dtxt ≪ "\n";

#define DV(v1) #v1 ≪ "  =  " ≪ v1
#define DV2(v1,v2) #v1 ≪ "  =  " ≪ v1 ≪ ",  " ≪ #v2 ≪ "  =  " ≪ v2
#define DV3(v1,v2,v3) #v1 ≪ "  =  " ≪ v1 ≪ ",  " ≪ #v2 ≪ "  =  " ≪ v2 \
                  ≪ ",  " ≪ #v3 ≪ "  =  " ≪ v3


// Use for debugging like this: DO_DEBUG() { debugging_code_here(); }

#define DO_DEBUG(num) if (num < DEBUG_LVL) /*nop*/; else

#ifdef __GNUG__
# ifndef NDEBUG
// extern "C" void volatile exit(int);

# define _assert(ex) {if (!(ex)){cerr ≪ "Assertion - " ≪ #ex ≪ " - failed:
file \"" ≪ __FILE__ ≪ "\", line " ≪ __LINE__ ≪ "\n";exit(1);}}
# define assert(ex) _assert(ex)
# else
# define _assert(ex)
# define assert(ex)
```

```
# define DEBOUT(num,dtxt)
# define DEBVAR(num,dtxt)
# define DV(v1)
# define DV2(v1,v2)
# define DV3(v1,v2,v3)
# endif /* NDEBUG */
#else
# include <assert.h>
#endif /* __GNUG__ */

// InternErr() can't be a function: we want to pick up __FILE__ and __LINE__

#define InternErr()\
   cerr << "Internal error, \"" << __FILE__ << "\", line " << __LINE__ <<
"\n"

#endif /* _main_h */
```

enum.h

```
/* -*- C++ -*- $Id: $ */

#ifndef _enum_h
#define _enum_h

#include "main.h"

#define MAXX 10 /* the max Geometry points horizontally */
#define MAXY 10 /* the max Geometry points vertically */

#define POINTS 9 /* the number of points in a 3x3 array */

#define NAME_LENGTH 100 /* max name length */
#define MAX_TERMS 8 /* the max number of Terms allowed */
#define SD_CONN 10 /* max source or drain connections to Nets */
#define GATE_CONN 10 /* max gate connections to Nets */
#define IO_CONN 10 /* max I/O connections to Nets */

enum El_type {
  TRANS_SLEFT, // transistor source left

  TRANS_SRIGHT, // transistor source right

  TRANS_SUP, // transistor source up

  TRANS_SDOWN // transistor source down

};

// @@ Convert to an enum


#define ELEMENTS 11 /* number of different Geometry elements */

#define WIRE_DIFF 4 /* wire in diffusion */
#define CONTACT_DIFF_M1 5 /* contact between diffusion and metal1 */
#define WIRE_POLY 6 /* wire in polysilicon */
#define CONTACT_POLY_M1 7 /* contact between polysilicon and metal1 */
#define VIA 8 /* contact between metal1 and metal2 */
#define WIRE_M1 9 /* wire in metal1 */
#define WIRE_M2 10 /* wire in metal2 */


enum Dir { // Direction

  NORTH,
```

```
  SOUTH,
  EAST,
  WEST
};


/* the following definitions are used in the technology file to
    specify legality of placements according to designrules, and to
    specify connectivity of the different objects relative to one another
*/

enum Legality {
  NO_EFFECT, // geometry object is legal to place, nodes
                                        // in the geometry are not connected, and
                                        // can't be, without further geometry objects
  CONNECTED, // legal placement, connected nodes
  ILLEGAL, // illegal placement according to designrules
  CAN_CONNECT, // legal, node may or may not be connected
  CONN_S, // legal, transistor source node connected
  CONN_D, // legal, transistor drain node connected
  CONN_G, // legal, transistor gate node connected
  CONN_S_DIR, // legal, source node directly connected
                                        // "between" geometry grid points
  CONN_D_DIR, // legal, drain node directly connected
  CAN_CONN_S, // legal, source node may or may not be conn.
  CAN_CONN_D, // legal, drain node may or may not be conn.
  CAN_CONN_G // legal, gate node may or may not be conn.
};

extern int element_connect[ELEMENTS][ELEMENTS][POINTS];

// @@ Should really be in another file?
```

extern int colmax; // *Actual max (instead of MAXX)*

extern int rowmax; // *Actual max (instead of MAXY)*

extern int transmax; // *Number of transistors to place*

`#endif /* _enum_h */`

pos.h

```
/* -*- C++ -*- $Id: pos_elem.h,v 1.7 1991/10/11 14:58:50 dash Exp $ */

#ifndef _pos_h
#define _pos_h

#include "main.h"

// This is the pure abstract class Pos - positions (must subclass).

// Used by try_next().

class Pos {
public:
   virtual Pos* next() = 0; // Next position

   virtual void first_pos() = 0; // Init for iteration

   virtual void operator++() = 0; // Iterator

   virtual int legal_pos() const = 0; // Senteniel for iteration.

};

#endif /* _pos_h */
```

elem.h

```
/* -*- C++ -*- $Id: pos_elem.h,v 1.7 1991/10/11 14:58:50 dash Exp $ */

#ifndef _elem_h
#define _elem_h

#include "main.h"

// This is the pure abstract class Elem - elements (must subclass).

// Used by try_next().

class Elem {
public:
  virtual Elem* next() = 0; // Next element

  virtual int more_elem() const = 0; // Senteniel for iteration.

};

#endif /* _elem_h */
```

list.h

```
/* -*- C++ -*- $Id: list.h,v 1.2 1991/08/12 01:06:58 dash Exp $ */

#ifndef _list_h
#define _list_h

#include "main.h"


#define ACTIVE 0x05AF /* bit-pattern for valid List objects */

class List {
public:
  List();
  List(List*);
  ~List();
  int OK(); // Checks the representation invariant


protected:
  int id; // Valid value: ACTIVE, invalid: anything else

  List *next; // Next list object, or NULL

};

#endif /* _list_h */
```

list.cc

```
/* $Id: list.cc,v 1.1 1991/08/12 00:53:42 dash Exp $ */


#include "main.h"
#include "list.h"

// ************************************

List::List()
{
  id = ACTIVE; // Make this a legal object

  next = NULL; // Make next a legal object too

}

List::List(List* l)
{
  id = ACTIVE; // Make this a legal object

  assert(l == NULL || l->OK());
  next = l; // Make next a legal object too

}

List::~List()
{
  assert(id == ACTIVE);
  id = 0; // Make this an illegal object

}

// Do some test to see if this looks like a valid list

// Catches some trivial errors that otherwise would be hard to find

int List::OK()
{
  if (id != ACTIVE) {
    cerr << "List error, id= " << id << "\n";
    return(FALSE);
  }
  if (next != NULL && next->id != ACTIVE) {
    cerr << "List error, next->id= " << next->id << "\n";
    return(FALSE);
  }
```

```
    return(TRUE);
}
```

undo.h

```
/* -*- C++ -*- $Id: backtrack.h,v 1.7 1991/10/10 18:11:06 dash Exp $ */

#ifndef _undo_h
#define _undo_h

#include "main.h"
#include "list.h"

// These classes are designed to reduce programming time, not primarily for

// execution time efficiency


// Recursively updating a global datastructure demands that all changes are

// noted and old values are reset during backtracking. Therefore all old

// values must be saved. Only a few types are handled so far, but any type can

// be supported by overloading.


// "Undo Saver" must be defined locally whenever a new "context" is entered.

// The Saver must be passed on to all routines called if the routines may

// change the global datastructure.


// Example of use of the Saver in action:

//

// void exampleroutine() {

// Undo Saver;

// var1 = 1;

// Saver[var1] = -1;

// .

// .

// Saver.restore(); // var1 is set back to 1
```

```
// }


// Each type known to the Undo-system must have its unique number

enum Typetag {
   INT, // Int

   INTP, // Int pointer, etc

   VOIDP,
   CHAR
};

class Bind : public List {
public:
   Bind(Bind*, int *);
   Bind(Bind*, int** );
   Bind(Bind*, void** );
   Bind(Bind*, char *);
   ~Bind();

private:
   void *var; // pointer to variable

   Typetag typetag; // CHAR, INT, INTP, osv.

   union {
      int as_int;
      int* as_intp;
      void* as_voidp;
      char as_char;
   } value;
   friend class Undo;
};

class Undo {
public:
   Undo();
   Undo& operator[](int&); // Remember address and old value

   void operator=(int); // New value to int pointed to in first object

   Undo& operator[](int*); // Similarly with these...

   void operator=(int*);
   Undo& operator[](void*&);
```

```
    void operator=(void*);
    Undo& operator[](char&);
    void operator=(char);
    void restore();

private:
    Bind* bindings; // Variable - Value list

};

#endif /* _undo_h */
```

undo.cc

```
/* $Id: backtrack.cc,v 1.9 1991/10/10 18:11:02 dash Exp $ */


#include "main.h"
#include "undo.h"

//**************************************

// Constructor for Bind

// Params: next, pointer to variable

Bind::Bind(Bind* n, int *variable)
{
  next = n;
  var = variable;
  value.as_int = *variable;
  typetag = INT;
  DEBOUT(1, "Adresse, Verdi:  " << (int) var << ",  " << value.as_int);
}


// Constructor for Bind

// Params: next, pointer to variable

Bind::Bind(Bind* n, int* *variable)
{
  next = n;
  var = variable;
  value.as_intp = *variable;
  typetag = INTP;
}


// Constructor for Bind

// Params: next, pointer to variable

Bind::Bind(Bind* n, void* *variable)
{
  next = n;
  var = variable;
  value.as_voidp = *variable;
  typetag = VOIDP;
}
```

```
// Constructor for Bind

// Params: next, pointer to variable

Bind::Bind(Bind* n, char *variable)
{
  next = n;
  var = variable;
  value.as_char = *variable;
  typetag = CHAR;
}


// Destructor for Bind

Bind::~Bind()
{
  DEBOUT(1, "~Bind: Adresse, Verdi: " << (int) var << ", " <<
value.as_int);
  switch (typetag) {
  case INT:
    *((int *) var) = value.as_int;
    break;
  case INTP:
    *((int** ) var) = value.as_intp;
    break;
  case VOIDP:
    *((void** ) var) = value.as_voidp;
    break;
  case CHAR:
    *((char *) var) = value.as_char;
    break;
  default:
    InternErr();
    break;
  }
}

// ************************************

// Constructor

Undo::Undo()
{
  bindings = NULL;
}
```

```
// Maybe I should have a macro to expand these?

#define UNDO_TYPE(Type) \
Undo& Undo::operator[](Type& variable) \
{ DEBOUT(1, "[]:  Adresse:  " << (int) &variable); \
  bindings = new Bind(bindings, &variable); \
  return *this;} \
void Undo::operator=(Type expr) \
{ assert(bindings->OK()); \
  if (bindings->typetag == INT) *((Type *) bindings->var) = expr; \
  else cerr << "Expected a " << #Type << \
    " now. Maybe you need an explisitt cast?\n"; \
}


// *** INT:


// Remember address and old value of the int

Undo& Undo::operator[](int& variable)
{
  DEBOUT(1, "[]:  Adresse:  " << (int) &variable);
  bindings = new Bind(bindings, &variable); // Put new object in front

  return *this;
}

// New value to int pointed to in first object

void Undo::operator=(int expr)
{
  assert(bindings->OK());
  if (bindings->typetag == INT)
    *((int *) bindings->var) = expr; // Set variable to value

  else
    cerr << "Expected an int now. Maybe you need an explisitt
cast?\n";
}

// *** INT*:

// OBS: Because of an error in C++, this is not declared as

// operator[](int* & variable), but as operator[](int* variable)
```

```
// This means that one have to pass a pointer in the place of use


// Remember address and old value of the int*

Undo& Undo::operator[](int* variable)
{
  DEBOUT(1, "[]: Adresse: " << (int) &variable);
  bindings = new Bind(bindings, variable); // Put new object in front

  return *this;
}

// New value to int-pointer pointed to in first object

void Undo::operator=(int* expr)
{
  assert(bindings->OK());
  if (bindings->typetag == INTP)
    *((int* *) bindings->var) = expr; // Set variable to value

  else
    cerr << "Expected an int* now. Maybe you need an explisitt
cast?\n";
}

// *** VOID*:


// Remember address and old value of the void*

Undo& Undo::operator[](void* & variable)
{
  bindings = new Bind(bindings, &variable); // Put new object in front

  return *this;
}

// New value to void-pointer pointed to in first object

void Undo::operator=(void* expr)
{
  assert(bindings->OK());
  if (bindings->typetag == VOIDP)
    *((void* *) bindings->var) = expr; // Set variable to value

  else
    cerr << "Expected an void* now. Maybe you need an explisitt
```

```
cast?\n";
}

// *** CHAR:


// Remember address and old value of the int*

Undo& Undo::operator[](char& variable)
{
  bindings = new Bind(bindings, &variable); // Put new object in front

  return *this;
}

// New value to char pointed to in first object

void Undo::operator=(char expr)
{
  assert(bindings->OK());
  if (bindings->typetag == CHAR)
    *((char *) bindings->var) = expr; // Set variable to value

  else
    cerr << "Expected an char now. Maybe you need an explisitt
cast?\n";
}

// This is the general restore routine

void Undo::restore()
{
  Bind* tmp; // Temp to hold Bind'ing to be deleted

  int debug_i = 0; // Only for debugging


  while (bindings != NULL) {
    DO_DEBUG(2) {debug_i++;}
    tmp = bindings;
    bindings = (Bind*) bindings->next;
    delete tmp; // delete Bind object in front

  }
  DEBOUT(2, "I restorel|kka " << debug_i << " ganger");
}
```

try_next.h

```
/* -*- C++ -*- $Id: $ */

#ifndef _try_next_h
#define _try_next_h

#include "main.h"

class Pos;
class Elem;

extern void try_next(Pos&, Elem&, void (*)());

#endif /* _try_next_h */
```

try_next.cc

```
/* $Id: $ */

#include "main.h"
#include "try_next.h"
#include "undo.h"
#include "placebounds.h"
#include "pos.h"
#include "elem.h"

extern int num_try_next;

// Loop through the tests in the bounds array

// Returns FALSE immediately if any test returns FALSE, else return TRUE

static int andtests(Pos & p, Elem & e, Undo& Saver)
{
  for (int tst = 0; bound[tst] ≠ NULL; tst++) // Try all bounds-tests

    if (!(*bound[tst])(p, e, Saver))
      return FALSE;
  return TRUE;
}


// This the main driver routine in the Branch And Bound approach.

// It is independant of the actual data it operates on.

// Just call it with proper instanses of Elem and Pos subclasses.

void try_next(Pos &p, Elem &e, void (*finished)())
{
  Undo Saver;

  for (; p.legal_pos(); p++) { // For all posistions...

    num_try_next++;
    if (andtests(p, e, Saver)) { // Try next pos if FALSE

      if (e.more_elem()) // Last Elem?

        try_next(*p.next(), *e.next(), finished); // ... then place next

      else
        finished(); // All elements placed successfully
```

```
        }
        Saver.restore(); // Tried all positions, remove placement

    }
}
```

gpos.h

```
/* -*- C++ -*- $Id: tpos_trans.h,v 1.12 1991/10/11 14:58:57 dash Exp $ */

#ifndef _gpos_h
#define _gpos_h

#include "main.h"
#include "pos.h"
#include "enum.h"

// ************************

// This is the grid positions

class Gpos : public Pos {
public:
  int x; // x coordinate

  int y; // y coordinate


  Gpos(); // Constructor

  Gpos(int xx, int yy); // Constructor

  Gpos& operator=(const Gpos&);
  virtual Pos* next(); // Next position

  virtual void first_pos(); // Init for iteration

  virtual void operator++(); // Iterator

  virtual int legal_pos() const;// End condition of iteration

  Gpos& north_pos();
  Gpos& south_pos();
  Gpos& east_pos();
  Gpos& west_pos();
};

// ************************

// This is an iterator for grid positions => Gpos with extra state info

// Returns the 4 neighbors in sequence, when ++ is used on object

class N4pos : public Gpos {
```

```
public:
   int nn; // Neighbor number


   N4pos(Gpos&); // Constructor

   virtual void operator++(); // Iterates to next neighbor

   virtual int l_pos(); // End condition of iteration

};

// ************************

// This is an iterator for grid positions => Gpos with extra state info

// Returns the 8 neighbors in sequence, when ++ is used on object

class N8pos : public Gpos {
public:
   int nn; // Neighbor number


   N8pos(Gpos&); // Constructor

   virtual void operator++(); // Iterates to next neighbor

   virtual int legal_pos() const;// End condition of iteration

};

#endif /* _gpos_h */
```

gpos.cc

```
/* $Id: tpos_trans.cc,v 1.16 1991/10/11 14:58:53 dash Exp $ */


#include "main.h"
#include "gpos.h"
#include "vlsi.h"

//************************

// Constructor for Grid-position objects

Gpos::Gpos()
{
   x = 1;
   y = 1;
}

// Constructor for Grid-position objects

Gpos::Gpos(int xx, int yy)
{
   x = xx;
   y = yy;
}

Gpos& Gpos::operator=(const Gpos& other)
{
   x = other.x;
   y = other.y;
   return *this;
}

// Next Gpos from this

Pos* Gpos::next()
{
   return new Gpos();
}


void Gpos::first_pos()
{
   x = 1;
   y = 1;
}
```

```
// Iterator for Gpos

void Gpos::operator++()
{
   if (colmax ≤ ++x) {
      x = 1;
      y++;
   }
}

// @@ Make this a operator int()?

// Test order trimmed for effeciency

int Gpos::legal_pos() const
{
   return (y < rowmax && 0 < x && x < colmax && 0 < y);
}


Gpos& Gpos::north_pos()
{
   Gpos *ret = new Gpos();

   ret→x = x;
   ret→y = y+1;

   return (*ret);
}


Gpos& Gpos::south_pos()
{
   Gpos *ret = new Gpos();

   ret→x = x;
   ret→y = y-1;


   return (*ret);
}


Gpos& Gpos::east_pos()
{
   Gpos *ret = new Gpos();

   ret→x = x+1;
```

```
    ret→y = y;

    return (*ret);
}


Gpos& Gpos::west_pos()
{
    Gpos *ret = new Gpos();

    ret→x = x-1;
    ret→y = y;

    return (*ret);
}

// ************************

// Constructor for Neighbor-position objects

N4pos::N4pos(Gpos& gp)
{
    assert(gp.legal_pos());

    x = gp.x;
    y = gp.y-1;
    nn = 0;
}

// Next neighbor

void N4pos::operator++()
{
    switch (++nn) { // Advance neighbor number

    case 1:
        x++;
        y++;
        break;
    case 2:
        x--;
        y++;
        break;
    case 3:
        x--;
        y--;
        break;
    case 4:
```

```
      break;
   default:
      InternErr();
      break;
   }
}

// Function to be called to find out if we can end the iteration

int N4pos::l_pos()
{
   DEBOUT(1, "Nabo-nummer, Sannhetsverdi (nn < 4): " ≪ nn ≪ ", "
≪ (nn < 4));
   return (nn < 4);
}


// ************************

// Constructor for Neighbor-position objects

N8pos::N8pos(Gpos& gp)
{
   assert(gp.legal_pos());

   x = gp.x-1;
   y = gp.y-1;
   nn = 0;
}

// Next neighbor

void N8pos::operator++()
{
   nn++; // Advance neighbor number

   if (nn == 4) { // Skip center point

      nn++;
      x++;
   }
   if (nn==3 || nn==6) {
      x = x-2;
      y++;
   }
   else
      x++;
}
```

```
// Function to be called to find out if we can end the iteration

int N8pos::legal_pos() const
{
   return (nn < 9);
}
```

tpos.h

```
/* -*- C++ -*- $Id: tpos_trans.h,v 1.12 1991/10/11 14:58:57 dash Exp $ */

#ifndef _tpos_h
#define _tpos_h

#include "main.h"
#include "gpos.h"
#include "enum.h"

// ************************

// This is the transistor positions

class Tpos : public Gpos {
public:
  El_type direction; // Direction of transistor


  Tpos(); // Constructor

  virtual Pos* next(); // Next position

  virtual void first_pos(); // Init for iteration

  virtual void operator++(); // Iterator

  int diff_outside_grid(); // Did diff land outside the grid

  Gpos& source_pos();
  Gpos& drain_pos();
  Gpos& gate1_pos();
  Gpos& gate2_pos();
};

#endif /* _tpos.h */
```

tpos.cc

```
/* $Id: tpos_trans.cc,v 1.16 1991/10/11 14:58:53 dash Exp $ */


#include "main.h"
#include "tpos.h"
#include "vlsi.h"

// ************************

// Constructor for Transistor-position objects

Tpos::Tpos()
{
   direction = 0;
}

// Next Tpos from this, Used in recursion

Pos* Tpos::next()
{
   return new Tpos();
}

void Tpos::first_pos()
{
   direction = 0;
}

// Iterator for Tpos

void Tpos::operator++()
{
   direction++;
   if (4 <= direction) {
      direction = 0;
      x++;
      if (colmax <= x) {
         x = 1;
         y++;
      }
   }
}


// ************************************
```

```
// Returns TRUE if we now are about to place diff outside internal grid area,

// else return FALSE

int Tpos::diff_outside_grid()
{
   return (x==1 && (direction==TRANS_SLEFT || direction == TRANS_SRIGHT)) ||
         (x==colmax-1 && (direction==TRANS_SLEFT || direction == TRANS_SRIGHT))
||
         (y==1 && (direction == TRANS_SUP || direction == TRANS_SDOWN)) ||
         (y==rowmax-1 && (direction == TRANS_SUP || direction == TRANS_SDOWN));
}

Gpos &Tpos::source_pos()
{
   Gpos *ret = new Tpos();

   ret->x = x;
   ret->y = y;

   switch (direction) {
   case TRANS_SLEFT:
      (ret->x)--;
      break;
   case TRANS_SRIGHT:
      (ret->x)++;
      break;
   case TRANS_SUP:
      (ret->y)++;
      break;
   case TRANS_SDOWN:
      (ret->y)--;
      break;
   default:
      InternErr();
   }

   return (*ret);
}


Gpos &Tpos::drain_pos()
{
   Gpos *ret = new Tpos();

   ret->x = x;
   ret->y = y;
```

```
    switch (direction) {
    case TRANS_SLEFT:
        (ret→x)++;
        break;
    case TRANS_SRIGHT:
        (ret→x)--;
        break;
    case TRANS_SUP:
        (ret→y)--;
        break;
    case TRANS_SDOWN:
        (ret→y)++;
        break;
    default:
        InternErr();
    }

    return (*ret);
}


// Definition: Gate1 is the one to the right when source is away.

Gpos &Tpos::gate1_pos()
{
    Gpos *ret = new Tpos();

    ret→x = x;
    ret→y = y;

    switch (direction) {
    case TRANS_SLEFT:
        (ret→y)++;
        break;
    case TRANS_SRIGHT:
        (ret→y)--;
        break;
    case TRANS_SUP:
        (ret→x)++;
        break;
    case TRANS_SDOWN:
        (ret→x)--;
        break;
    default:
        InternErr();
    }

    return (*ret);
```

```
}


// Definition: Gate2 is the one to the left when source is away.

Gpos &Tpos::gate2_pos()
{
   Gpos *ret = new Tpos();

   ret->x = x;
   ret->y = y;

   switch (direction) {
   case TRANS_SLEFT:
      (ret->y)--;
      break;
   case TRANS_SRIGHT:
      (ret->y)++;
      break;
   case TRANS_SUP:
      (ret->x)--;
      break;
   case TRANS_SDOWN:
      (ret->x)++;
      break;
   default:
      InternErr();
   }

   return (*ret);
}
```

trans.h

```
/* -*- C++ -*- $Id: tpos_trans.h,v 1.12 1991/10/11 14:58:57 dash Exp $ */

#ifndef _trans_h
#define _trans_h

#include "main.h"
#include "elem.h"
#include "enum.h"

class Terminal; // Needed by the class Trans

class Gpos;
class Geometry;

// ************************

// This is the transistors

class Trans : public Elem {
public:
   int ident; // Unique identifier for each Trans

   Geometry* geomp; // Pointer to geometry-point

// El_type direction; // @@@ Also direction. Into geometry?

   Terminal *source;
   Terminal *drain;
   Terminal *gate;

   Trans(); // Constructor

   Trans(int nr); // Constructor

   virtual Elem* next(); // Next element

   virtual int more_elem() const; // End condition of iteration

   Trans& operator=(const Trans&);
   friend int operator==(const Trans&, const Trans&);
   int OK(); // Checks the representation invariant

};

#endif /* _trans.h */
```

trans.cc

```
/* $Id: tpos_trans.cc,v 1.16 1991/10/11 14:58:53 dash Exp $ */


#include "main.h"
#include "trans.h"
#include "vlsi.h"
#include "geometry.h"

// ************************************

// Constructor for Trans objects

Trans::Trans()
{
  DEBOUT(1, "new Trans()");
  ident = 0;
  geomp = 0;
}

// Constructor for Trans objects

Trans::Trans(int nr)
{
  DEBOUT(1, "new Trans(" << nr << ")");
  ident = nr;
  geomp = 0;
}

// Next Trans from this

Elem* Trans::next()
{
  assert(this->more_elem());
  return trans_array[ident+1];
}

int Trans::more_elem() const
{
  return (ident != transmax-1);
}

Trans& Trans::operator=(const Trans& other)
{
  ident = other.ident;
  return *this;
}
```

```
int operator==(const Trans& t1, const Trans& t2)
{
   return (t1.ident == t2.ident);
}


// Checks the representation invariant

int Trans::OK()
{
   return TRUE;
}
```

io.h

```
/* -*- C++ -*- $Id: $ */

#ifndef _io_h
#define _io_h

#include "main.h"
#include "enum.h"

class Net;
class Terminal;

// ****************************************


// Class describes the Io connections

class Io {
public:
  Net* n; // Net this Io is member of

  Terminal *term;
  int layer; // WIRE_M1, ...

  int direct; // NORTH, SOUTH, EAST, WEST

  int offset; // A number, or '-1' meaning "floating"


  Io(Net*, int, int, int);

  int OK(); // Checks the representation invariant

};

#endif /* _io_h */
```

io.cc

```
/* $Id: $ */


#include "main.h"
#include "io.h"

// ************************************


Io::Io(Net* netp, int elem_nr, int edge, int float_num) {
   n = netp;
   layer = elem_nr;
   direct = edge;
   offset = float_num;
}

int Io::OK() {
   return TRUE;
}
```

vlsi.h

```
/* -*- C++ -*- $Id: vlsi.h,v 1.19 1991/10/11 14:59:04 dash Exp $ */

#ifndef _vlsi_h
#define _vlsi_h

#include "main.h"
#include "grid.h"
#include "enum.h"

class Net; // So that we can refer to Net...

class Undo; // ... Undo ...

class Trans; // ... and Trans from now on


extern Trans* trans_array[1000]; // @@@ transmax

extern Net* net_array[1000]; // @@@ ??

extern Grid grid;

extern void read_netlist(char* fname);
extern void write_mighty_input(char* fname, Undo&);
extern void read_mighty_output(char* fname, Undo&);

#endif /* _vlsi_h */
```

vlsi.cc

```
/* $Id: vlsi.cc,v 1.15 1991/10/11 14:58:59 dash Exp $ */


#include "main.h"
#include "vlsi.h"
#include "tpos.h"
#include "trans.h"
#include "geometry.h"
#include "net.h"
#include "io.h"
#include "terminal.h"
#include "undo.h"
#include "placebounds.h"

#include "tech.incl" /* Special, not really a header file */

#include <stdio.h>
#include <string.h>

// extern "C" { int strlen(const char*); }


int colmax; // Actual max (instead of MAXX)

int rowmax; // Actual max (instead of MAXY)

int transmax; // Number of transistors to place

int maxnet_nr; // Number of nets actually used by in-file

int io_nr = 0;

Trans* trans_array[1000]; // @@ transmax?

Net* net_array[1000]; // @@ maxnet_nr?

Io* io_array[1000]; // @@ maxio_nr?

Grid grid;


// set all array positions to NULL

void init_arrays()
{
   for(int x=0; x<=1000; x++) {
```

```
      trans_array[x] = NULL;
      net_array[x] = NULL;
   }
}


void read_netlist(char* fname)
{
   filebuf* f1 = new filebuf(); // Filebuf used by the istream from

   int source_nr; // Net-nr of the source Terminal of Trans

   int drain_nr; // Net-nr of the drain Terminal

   int gate_nr; // Net-nr of the gate Terminal

   int trans_nr = 0; // The number of Trans seen so far

   int end = FALSE; //

   char fc; //


   if (f1->open(fname, input) == 0) {
      cerr << "cannot open input file " << fname << "\n";
      exit(1);
   }
   istream from(f1);

   maxnet_nr = -1;

   for (int line = 0; trans_nr < transmax && !end && cin; line++) {
      from >> fc;
      switch (fc) {
      case '#': // Skip until another #

         from >> fc;
         while (fc != '#') {
            from >> fc;
         }
         break;
      case '@':
         end = TRUE;
         cerr << "To few T-lines in input file " << fname << " to
support "
            << transmax << " transistors.\n";
         exit(1);
         break;
```

```
    case 'T':
    case 't':
      from >> source_nr >> drain_nr >> gate_nr;

      maxnet_nr = max(source_nr, maxnet_nr);
      maxnet_nr = max(drain_nr, maxnet_nr);
      maxnet_nr = max(gate_nr, maxnet_nr);
      DEBOUT(2, source_nr << ", " << drain_nr << ", " << gate_nr);

      trans_array[trans_nr] = new Trans(trans_nr);

      if (net_array[source_nr] == NULL) // @@@ Make this a proc?

        net_array[source_nr] = new Net(source_nr);
      trans_array[trans_nr]->source = new Terminal(net_array[source_nr]);
      net_array[source_nr]->include(trans_array[trans_nr]->source);
      trans_array[trans_nr]->source->trans = trans_array[trans_nr];

      if (net_array[drain_nr] == NULL) // Similarily with drain...

        net_array[drain_nr] = new Net(drain_nr);
      trans_array[trans_nr]->drain = new Terminal(net_array[drain_nr]);
      net_array[drain_nr]->include(trans_array[trans_nr]->drain);
      trans_array[trans_nr]->drain->trans = trans_array[trans_nr];

      if (net_array[gate_nr] == NULL) // Similarily with gate...

        net_array[gate_nr] = new Net(gate_nr);
      trans_array[trans_nr]->gate = new Terminal(net_array[gate_nr]);
      net_array[gate_nr]->include(trans_array[trans_nr]->gate);
      trans_array[trans_nr]->gate->trans = trans_array[trans_nr];

      trans_nr++;
      break;
    case 'N':
    case 'n':
      int net_nr; // The Net-nr read when naming Nets

      from >> net_nr;
      if (net_array[net_nr] == NULL)
        net_array[net_nr] = new Net(net_nr);
      else
        maxnet_nr = max(net_nr, maxnet_nr);
      from >> net_array[net_nr]->name;
      assert(strlen(net_array[net_nr]->name) < NAME_LENGTH);
      break;
    case 'C':
    case 'c':
```

```
        int layer; // Which layer the contact is in

        int edge; // Edge: 1=north, 2=south, 3=east, 4=west

        int offset; // A number, or '-1' meaning "floating"

        from >> net_nr;
        from >> layer;
        from >> edge;
        from >> offset;
        maxnet_nr = max(net_nr, maxnet_nr);

        if (net_array[net_nr] == NULL)
          net_array[net_nr] = new Net(net_nr);
        DEBVAR(1, net_array[net_nr]->net_index);
        io_array[io_nr] = new Io(net_array[net_nr], layer, edge, offset);
        io_array[io_nr]->term = new Terminal(net_array[net_nr]);
        net_array[net_nr]->include(io_array[io_nr]->term);
        DEBVAR(1, io_nr);
        DEBVAR(1, io_array[io_nr]->n->net_index);
        io_nr++;
        break;
      default:
        cerr << "Error in line " << line << " in file " << fname << "\n";
        exit(1);
        break;
      }
    }
    f1->close();
    delete f1;
    DEBOUT(1, "<read_netlist()");
    DEBVAR(1, io_nr);

    assert(maxnet_nr < 1000);
    DO_DEBUG(111) {
      for (int count = 1; count < maxnet_nr; count++)
        assert(net_array[count] != 0);
    }
}


void write_mighty_input(char* fname, Undo& SC)
{
    FILE* f;
    Trans* t;
    int io_right = 0; // Number of Io-connections to the right

    int io_left = 0; // Number of Io-connections to the left
```

```
if ((f = fopen(fname, "w")) == 0) {
   cerr ≪ "cannot open output file " ≪ fname ≪ "\n";
   exit(1);
}

fprintf(f, "number_of_nets %d\n", maxnet_nr);
fprintf(f, "rectagoncorners 4\n");
fprintf(f, "0 0\n");
fprintf(f, "%d 0\n", colmax);
fprintf(f, "%d %d\n", colmax, rowmax);
fprintf(f, "0 %d\n", rowmax);
fprintf(f, "number_of_pins %d\n", transmax∗3);

for (Gpos gp; gp.legal_pos(); gp++) { // Really: Go through trans_array[]

   if (grid[gp].nr_elems ≠ 0) {
      for (int eli = 0; eli < grid[gp].nr_elems; eli++) {
         if (grid[gp].elements[eli] < 4) { // @@ ikke ha konstant her

            t = grid[gp].trns;
            switch (grid[gp].trans_dir) {
            case TRANS_SLEFT:
               place_extra_in(gp.east_pos(), WEST, WIRE_DIFF, SC);
               place_extra_in(gp.west_pos(), EAST, WIRE_DIFF, SC);
               fprintf(f, "%d %d %d 1\n", t→source→np→net_index, gp.x-1, gp.y);
               fprintf(f, "%d %d %d 1\n", t→gate→np→net_index, gp.x, gp.y);
               fprintf(f, "%d %d %d 1\n", t→drain→np→net_index, gp.x+1, gp.y);
               break;
            case TRANS_SRIGHT:
               place_extra_in(gp.east_pos(), WEST, WIRE_DIFF, SC);
               place_extra_in(gp.west_pos(), EAST, WIRE_DIFF, SC);
               fprintf(f, "%d %d %d 1\n", t→drain→np→net_index, gp.x-1, gp.y);
               fprintf(f, "%d %d %d 1\n", t→gate→np→net_index, gp.x, gp.y);
               fprintf(f, "%d %d %d 1\n", t→source→np→net_index, gp.x+1, gp.y);
               break;
            case TRANS_SUP:
               place_extra_in(gp.north_pos(), SOUTH, WIRE_DIFF, SC);
               place_extra_in(gp.south_pos(), NORTH, WIRE_DIFF, SC);
               fprintf(f, "%d %d %d 1\n", t→drain→np→net_index, gp.x, gp.y-1);
               fprintf(f, "%d %d %d 1\n", t→gate→np→net_index, gp.x, gp.y);
               fprintf(f, "%d %d %d 1\n", t→source→np→net_index, gp.x, gp.y+1);
               break;
            case TRANS_SDOWN:
               place_extra_in(gp.north_pos(), SOUTH, WIRE_DIFF, SC);
               place_extra_in(gp.south_pos(), NORTH, WIRE_DIFF, SC);
               fprintf(f, "%d %d %d 1\n", t→source→np→net_index, gp.x, gp.y-1);
```

```
                fprintf(f, "%d %d %d 1\n", t→gate→np→net_index, gp.x, gp.y);
                fprintf(f, "%d %d %d 1\n", t→drain→np→net_index, gp.x, gp.y+1);
                break;
            default:
                InternErr();
            }
        }
        else
            switch (grid[gp].elements[eli]) {
            case WIRE_DIFF:
            fprintf(f, "Wire %d, %d\n", gp.x, gp.y);
                break;
            default:
                InternErr();
            }
    }
  }
}

for (int i = 0; i < io_nr; i++) { // Count Io-connections

  if (io_array[i]→direct == WEST)
    io_right++;
  else
    if (io_array[i]→direct == EAST)
      io_left++;
}

if (0 < io_right) { // Do Io-connections to the right

  fprintf(f, "right_list %d\n", io_right);
  for (i = 0; i < io_nr; i++)
    if (io_array[i]→direct == WEST)
      fprintf(f, "%d ", io_array[i]→n→net_index);
  fprintf(f, "\n");
}

if (0 < io_left) { // Do Io-connections to the left

  fprintf(f, "left_list %d\n", io_left);
  for (i = 0; i < io_nr; i++) // Take care of Io-connections

    if (io_array[i]→direct == EAST)
      fprintf(f, "%d ", io_array[i]→n→net_index);
  fprintf(f, "\n");
}

fprintf(f, "obstacles %d\n", transmax); // Hack. Should really patch mighty
```

```
  for (gp.first_pos(); gp.legal_pos(); gp++) {
    if (grid[gp].nr_elems ≠ 0) {
      for (int eli = 0; eli < grid[gp].nr_elems; eli++) {
        if (grid[gp].elements[eli] < 4) { // @@ ikke ha konstant her

          fprintf(f, "%d %d %d %d 2\n", gp.x, gp.y, gp.x, gp.y);
        }
      }
    }
  }

  fclose(f);
}


void read_mighty_output(char* fname, Undo& SC)
{
  FILE* f;
  char txt[100];
  int nr_vias;
  int nr_wires;
  int netnr;
  int x1;
  int y1;
  int x2;
  int y2;
  int layer1;
  int layer2;
  int status = 0;
  int i;
  Gpos *gp;

  if ((f = fopen(fname, "r")) == 0) {
    cerr ≪ "cannot open input file " ≪ fname ≪ "\n";
    return;
  }

  while (status ≠ EOF && strcmp("vias", txt) ≠ 0) // Search to keyword 'vias'

    status = fscanf(f, "%s", txt);

  if (status ≠ 1) {
    cerr ≪ "Input file " ≪ fname ≪ " has wrong format.\n";
    cerr ≪ "Keyword 'vias' not found.\n";
    system("cat mighty.out");
    return;
```

```
}

status = fscanf(f, "%d\n", &nr_vias);

if (status ≠ 1) {
  cerr ≪ "Input file " ≪ fname ≪ " has wrong format.\n";
  cerr ≪ "No number found, expected number of vias.\n";
  system("cat mighty.out");
  return;
}

for (; 0 < nr_vias; nr_vias--) { // Reading in the vias

  status = fscanf(f, "%d %d %d %d %d", &netnr, &x1, &y1, &layer1, &layer2);

  if (status ≠ 5) {
    cerr ≪ "Input file " ≪ fname ≪ " has wrong format.\n";
    cerr ≪ "Expected list of vias, but no numbers found.\n";
    system("cat mighty.out");
    return;
  }
  gp = new Gpos(x1, y1);
  place_via(*gp, SC);
  delete gp;
}

status = fscanf(f, "%s", txt); // Next should be keyword "wires"


if (status ≠1 || strcmp("wires", txt) ≠ 0) {
  cerr ≪ "Input file " ≪ fname ≪ " has wrong format.\n";
  cerr ≪ "Keyword 'wires' not found.\n";
  system("cat mighty.out");
  return;
}

status = fscanf(f, "%d\n", &nr_wires);

if (status ≠ 1) {
  cerr ≪ "Input file " ≪ fname ≪ " has wrong format.\n";
  cerr ≪ "No number found, expected number of wires.\n";
  system("cat mighty.out");
  return;
}

for (; 0 < nr_wires; nr_wires--) { // Reading in the wires

  status = fscanf(f, "%d %d %d %d %d %d",
```

```
                    &netnr, &x1, &y1, &x2, &y2, &layer1);

  if (status ≠ 6) {
    cerr ≪ "Input file " ≪ fname ≪ " has wrong format.\n";
    cerr ≪ "Expected list of wires, but no numbers found.\n";
    system("cat mighty.out");
    return;
  }
  if (x1 == 0 || x1 == colmax)
    DEBOUT(111, "wire:   " ≪ x1 ≪ ",  " ≪ y1 );
  if (x2 == 0 || x2 == colmax)
    DEBOUT(111, "wire:   " ≪ x2 ≪ ",  " ≪ y2 );

  // Insert new data into grid

  if (x1 == x2) // The line is vertical

    for (i = min(y1, y2); i ≤ max(y1, y2); i++) {
      gp = new Gpos(x1, i);
      if (layer1 == 1) {

        if (grid[*gp].trns == 0) // If no trans present

          place_m1(*gp, SC);

        if (i ≠ min(y1, y2)) { // If not the start point

          place_extra_in(*gp, SOUTH, WIRE_M1, SC);
        }
        if (i ≠ max(y1, y2)) { // If not the end point

          place_extra_in(*gp, NORTH, WIRE_M1, SC);
        }
      }
      else {
        place_m2(*gp, SC);
        if (i ≠ min(y1, y2)) { // If not the start point

          place_extra_in(*gp, SOUTH, WIRE_M2, SC);
        }
        if (i ≠ max(y1, y2)) { // If not the end point

          place_extra_in(*gp, NORTH, WIRE_M2, SC);
        }
      }
      delete gp;
    }
  else // The line is horisontal
```

```
        for (i = min(x1, x2); i ≤ max(x1, x2); i++) {
            gp = new Gpos(i, y1);
            if (layer1 == 1) {
                if (grid[*gp].trns == 0) // If no trans present

                    place_m1(*gp, SC);
                if (i ≠ min(x1, x2)) { // If not the start point

                    place_extra_in(*gp, WEST, WIRE_M1, SC);
                }
                if (i ≠ max(x1, x2)) { // If not the end point

                    place_extra_in(*gp, EAST, WIRE_M1, SC);
                }
            }
            else {
                place_m2(*gp, SC);
                if (i ≠ min(x1, x2)) { // If not the start point

                    place_extra_in(*gp, WEST, WIRE_M2, SC);
                }
                if (i ≠ max(x1, x2)) { // If not the end point

                    place_extra_in(*gp, EAST, WIRE_M2, SC);
                }
            }
            delete gp;
        }
    }

    fclose(f);
}
```

ivtrans.h

```
/* -*- C++ -*- $Id: ivtrans.h,v 1.6 1991/10/07 22:33:25 dash Exp $ */

#ifndef _ivtrans_h
#define _ivtrans_h

#include "main.h"

#ifdef IV

#include <InterViews/interactor.h>

// By subclassing from Interactor we get access to much of InterViews

class Ivtrans : public Interactor {
public:
        Ivtrans();
        void Run();
        void ivout();

private:
        virtual void Reconfig();
        virtual void Redraw(Coord, Coord, Coord, Coord);
        virtual void Resize();
        virtual void Handle(Event &);
        void DrawTrans(Coord, Coord, int);
        void DrawDiff(Coord, Coord, Bool, Bool, Bool, Bool);
        void DrawPoly(Coord, Coord, Bool, Bool, Bool, Bool);
        void DrawM1(Coord, Coord, Bool, Bool, Bool, Bool);
        void DrawM2(Coord, Coord, Bool, Bool, Bool, Bool);
        void DrawVia(Coord, Coord);
        void DrawDiffM1(Coord, Coord);
        void DrawPolyM1(Coord, Coord);
        void CenterText(char*, Coord, Coord);
        void board_update();
        void board_print();
};

#endif /* IV */

#endif /* _ivtrans_h */
```

ivtrans.cc

```
/* $Id: ivtrans.cc,v 1.15 1991/10/11 14:58:40 dash Exp $ */

#include "main.h"

#ifdef IV

#include "ivtrans.h"
#include "tpos.h"
#include "trans.h"
#include "geometry.h"
#include "vlsi.h"
#include "undo.h"
#include "placebounds.h"
#include "try_next.h"

#include <InterViews/event.h>
#include <InterViews/painter.h>
#include <InterViews/pattern.h>
#include <InterViews/color.h>
#include <InterViews/sensor.h>
#include <InterViews/shape.h>
#include <InterViews/font.h>
#include <InterViews/world.h>
#include <InterViews/X11/Xlib.h>
#include <InterViews/X11/worldrep.h>

#include <stdio.h>
#include <stdlib.h>

const int width = 500; // Windows initial width

const int height = 500; // Windows initial height


// Short names on complex exprs - values are cached for efficiency

static Coord xr; // xradius

static Coord yr; // yradius

static Coord xdiam; // xdiameter

static Coord ydiam; // xdiameter


static int skip = 0; // Nr of solutions to generate before showing
```

```
static int refresh = FALSE; // Print the whole board, or just changes


static Event* e; // Event peker


static Pattern* stripesL;
static Pattern* stripesR;

static Color* m1color;
static Color* m2color;

extern int solve_nr; // Number of solutions found

extern Ivtrans* ivtrans;

Grid ivgrid;

// This function is called from try_next(), and is only a wrapper for

// ivtrans->ivout(). It doesn't work to have a pointer directly

// because pointer-to-member-function are a special type in C++,

// different from normal pointer-to-function.

void wrapper()
{
  ivtrans->ivout();
}


// Print out all of the board

void Ivtrans::board_print()
{
  char nr[20]; // At least enough space for what we need

  Coord xo, yo; // Short names on complex exprs


  output->ClearRect(canvas, 0, 0, xmax, ymax);

  for (Gpos gp; gp.legal_pos(); gp++) {
    xo = gp.x * xdiam;
    yo = gp.y * ydiam;
    Geometry& g = grid[gp];
```

```
if (g.nr_elems == 0)
  output→FillCircle(canvas, xo, yo, 4);
else {
  Bool layers[ELEMENTS] = // Sort the elements to make a nice drawing

    { FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE, FALSE, FALSE }; // Init layers-array all FALSE

  for (int eli = 0; eli < g.nr_elems; eli++) // TRUE for present elements

    layers[g.elements[eli]] = TRUE;

  for (int i = 0; i < ELEMENTS; i++) {
    if (layers[i]) { // If element i is present...

      switch (i) {
      case TRANS_SLEFT:
      case TRANS_SRIGHT:
      case TRANS_SUP:
      case TRANS_SDOWN:
        DrawTrans(xo, yo, g.trans_dir);
        sprintf(nr, "%d", g.trns→ident); // Convert to a string

        CenterText(nr, xo, yo);
        break;
      case WIRE_DIFF:
        DrawDiff(xo, yo, g.elem_dir[NORTH][i], g.elem_dir[SOUTH][i],
                 g.elem_dir[EAST][i], g.elem_dir[WEST][i]);
        break;
      case WIRE_POLY:
        DrawPoly(xo, yo, g.elem_dir[NORTH][i], g.elem_dir[SOUTH][i],
                 g.elem_dir[EAST][i], g.elem_dir[WEST][i]);
        break;
      case WIRE_M1:
        DrawM1(xo, yo, g.elem_dir[NORTH][i], g.elem_dir[SOUTH][i],
                 g.elem_dir[EAST][i], g.elem_dir[WEST][i]);
        break;
      case WIRE_M2:
        DrawM2(xo, yo, g.elem_dir[NORTH][i], g.elem_dir[SOUTH][i],
                 g.elem_dir[EAST][i], g.elem_dir[WEST][i]);
        break;
      case VIA:
        DrawVia(xo, yo);
        break;
      case CONTACT_DIFF_M1:
        DrawDiffM1(xo, yo);
        break;
      case CONTACT_POLY_M1:
```

```
                DrawPolyM1(xo, yo);
                break;
            default:
                InternErr();
            }
        }
    }
  }
  sprintf(nr, "Solution nr:  %d", solve_nr); // Convert solve_nr to a string

  output→Text(canvas, nr, 2, 2);
}

// Print out the board incrementally. Print only things that have changed

void Ivtrans::board_update()
{
  char nr[20]; // Overkill, but who cares?


  for (Gpos gp; gp.legal_pos(); gp++)
    if (!(grid[gp] == ivgrid[gp])) {
      Coord xo = gp.x * xdiam; // Cache the value

      Coord yo = gp.y * ydiam; // Cache the value

      if (grid[gp].nr_elems == 0) {
        output→ClearRect(canvas, xo-xr, yo-yr, xo+xr, yo+yr);
        output→FillCircle(canvas, xo, yo, 4);
      }
      else {
        if (ivgrid[gp].nr_elems ≠ 0)
          output→ClearRect(canvas, xo-xr, yo-yr, xo+xr, yo+yr);
        for (int eli = 0; eli < grid[gp].nr_elems; eli++) {
          if (grid[gp].elements[eli] < 4) {// @@ ikke ha konstant her

            DrawTrans(xo, yo, grid[gp].elements[eli]);
            sprintf(nr, "%d", grid[gp].trns→ident); // Convert to a string

            CenterText(nr, xo, yo);
          }
          else
            switch (grid[gp].elements[eli]) {
            case VIA: // @@@ Se board_print

              DrawVia(xo, yo);
              break;
```

```
                default:
                    InternErr();
                }
            }
        }
        ivgrid[gp] = grid[gp];
    }
  sprintf(nr, "Solution nr:  %d", solve_nr); // Convert solve_nr to a string

  output→Text(canvas, nr, 2, 2);
}


// Draw text centered around (xcoord, ycoord)

void Ivtrans::CenterText(char* txt, Coord xcoord, Coord ycoord)
{
  static Font* fnt = output→GetFont();
  static int fheight_2 = fnt→Height()/2;
  assert(fnt→Valid());

  output→Text(canvas, txt, xcoord-fnt→Width(txt)/2, ycoord-fheight_2);
}


// ** Section where we draw the various element types **


// Draw a transistor

void Ivtrans::DrawTrans(Coord xo, Coord yo, int dir)
{
  if (dir == TRANS_SLEFT || dir == TRANS_SRIGHT) {
    output→FillRect(canvas, xo-xr*7/10, yo-yr*3/10, xo+xr*7/10, yo+yr*3/10);
    output→ClearRect(canvas, xo-xr*2/10, yo-yr*6/10, xo+xr*2/10, yo+yr*6/10);
    output→Rect(canvas, xo-xr*2/10, yo-yr*6/10, xo+xr*2/10, yo+yr*6/10);
    if (dir == TRANS_SLEFT) {
      CenterText("S", xo-xr/2, yo);
      CenterText("D", xo+xr/2, yo);
    }
    else {
      CenterText("D", xo-xr/2, yo);
      CenterText("S", xo+xr/2, yo);
    }
  }
  else if (dir == TRANS_SUP || dir == TRANS_SDOWN) {
    output→FillRect(canvas, xo-xr*3/10, yo-yr*7/10, xo+xr*3/10, yo+yr*7/10);
    output→ClearRect(canvas, xo-xr*6/10, yo-yr*2/10, xo+xr*6/10, yo+yr*2/10);
```

```
      output→Rect(canvas, xo-xr*6/10, yo-yr*2/10, xo+xr*6/10, yo+yr*2/10);
      if (dir == TRANS_SDOWN) {
         CenterText("S", xo, yo-yr/2);
         CenterText("D", xo, yo+yr/2);
      }
      else {
         CenterText("D", xo, yo-yr/2);
         CenterText("S", xo, yo+yr/2);
      }
   }
   else InternErr();
}


// Draw diff

void Ivtrans::DrawDiff(Coord xo, Coord yo, Bool n, Bool s, Bool e, Bool w)
{
   output→FillRect(canvas, xo-xr/2, yo-yr/2, xo+xr/2, yo+yr/2);
   if (n) output→FillRect(canvas, xo-xr/2, yo+yr/2, xo+xr/2, yo+yr);
   if (s) output→FillRect(canvas, xo-xr/2, yo-yr, xo+xr/2, yo-yr/2);
   if (e) output→FillRect(canvas, xo+xr/2, yo-yr/2, xo+xr, yo+yr/2);
   if (w) output→FillRect(canvas, xo-xr, yo-yr/2, xo-xr/2, yo+yr/2);
   CenterText("D", xo, yo);
}

// Draw poly

void Ivtrans::DrawPoly(Coord xo, Coord yo, Bool n, Bool s, Bool e, Bool w)
{
   output→ClearRect(canvas, xo-xr/2, yo-yr/2, xo+xr/2, yo+yr/2);
   output→Rect(canvas, xo-xr/2, yo-yr/2, xo+xr/2, yo+yr/2);
   CenterText("P", xo, yo);
}

// Draw M1

void Ivtrans::DrawM1(Coord xo, Coord yo, Bool n, Bool s, Bool e, Bool w)
{
   output→SetPattern(stripesL);
   output→SetColors(m1color, nil);
   output→FillRect(canvas, xo-xr/2, yo-yr/2, xo+xr/2, yo+yr/2);
   if (n) output→FillRect(canvas, xo-xr/2, yo+yr/2, xo+xr/2, yo+yr);
   if (s) output→FillRect(canvas, xo-xr/2, yo-yr, xo+xr/2, yo-yr/2);
   if (e) output→FillRect(canvas, xo+xr/2, yo-yr/2, xo+xr, yo+yr/2);
   if (w) output→FillRect(canvas, xo-xr, yo-yr/2, xo-xr/2, yo+yr/2);
   output→SetColors(black, nil);
   output→SetPattern(solid);
```

```
    CenterText("M1", xo, yo);
}

// Draw M1

void Ivtrans::DrawM2(Coord xo, Coord yo, Bool n, Bool s, Bool e, Bool w)
{
    boolean origfill; // IV-version of Bool/boolean

    origfill = output->BgFilled();
    output->FillBg(false);
    output->SetPattern(stripesR);
    output->SetColors(m2color, nil);
//output->FillRect(canvas, xo-xr/4, yo-yr/4, xo+xr/4, yo+yr/4);

    output->FillRect(canvas, xo-xr/2, yo-yr/2, xo+xr/2, yo+yr/2);
    if (n) output->FillRect(canvas, xo-xr/2, yo+yr/2, xo+xr/2, yo+yr);
    if (s) output->FillRect(canvas, xo-xr/2, yo-yr, xo+xr/2, yo-yr/2);
    if (e) output->FillRect(canvas, xo+xr/2, yo-yr/2, xo+xr, yo+yr/2);
    if (w) output->FillRect(canvas, xo-xr, yo-yr/2, xo-xr/2, yo+yr/2);
    output->SetColors(black, nil);
    output->SetPattern(solid);
    output->FillBg(origfill);
    CenterText("M2", xo, yo);
}

// Draw a VIA between M1 and M2

void Ivtrans::DrawVia(Coord xo, Coord yo)
{
    output->SetPattern(darkgray);
    output->FillRect(canvas, xo-xr*2/3, yo-yr*2/3, xo+xr*2/3, yo+yr*2/3);
    CenterText("VIA", xo, yo);
    output->SetPattern(solid);
}


// Draw a VIA between Diff and M1

void Ivtrans::DrawDiffM1(Coord xo, Coord yo)
{
    output->SetPattern(darkgray);
    output->FillRect(canvas, xo-xr*2/3, yo-yr*2/3, xo+xr*2/3, yo+yr*2/3);
    CenterText("D-M1", xo, yo);
    output->SetPattern(solid);
}
```

```
// Draw a VIA between Poly and M1

void Ivtrans::DrawPolyM1(Coord xo, Coord yo)
{
   output→SetPattern(darkgray);
   output→FillRect(canvas, xo-xr*2/3, yo-yr*2/3, xo+xr*2/3, yo+yr*2/3);
   CenterText("P-M1", xo, yo);
   output→SetPattern(solid);
}



// Read events and print the board

void Ivtrans::ivout()
{
   solve_nr++; // Found one more solution


   if (skip > 1)
      skip--;
   else {
      if (refresh) {
         board_print(); // Print out the board

         refresh = FALSE;
      }
      else
         board_update(); // Print out changes only


      skip = 0;
      do {
         Read(*e); // Just hang until user presses a key/button

         e→target→Handle(*e);
      } while (skip==0); // Exit loop on first real keypress

   }
}



// Read events until user wants a solution

// Rest of Events-handling is done in ivout()

void Ivtrans::Run()
```

```
{
   ivgrid.init(colmax, rowmax); // Initialize grid

   stripesL = new Pattern(0x1248); // Interpreted as a 4x4 bitmap:

                                                  // 0001 = 1

                                                  // 0010 = 2

                                                  // 0100 = 4

                                                  // 1000 = 8

   stripesL→Reference();
   stripesR = new Pattern(0x8421); // Interpreted as a 4x4 bitmap:

                                                  // 1000 = 8

                                                  // 0100 = 4

                                                  // 0010 = 2

                                                  // 0001 = 1

   stripesR→Reference();

// The DefaultDepth(display, screen_number) macro returns the depth

// (number of planes) of the default root window for the specified

// screen.

// The DefaultScreen(display) macro returns the default screen number

// referenced in the XOpenDisplay routine.

   if (DefaultDepth(this→GetWorld()→Rep()→display(),
                         DefaultScreen(this→GetWorld()→Rep()→display())) == 1) {
      m1color = black;
      m2color = black;
   }
   else {
      m1color = new Color("blue");
      m1color→Reference();
      m2color = new Color("violet");
      m2color→Reference();
   }
```

```
    e = new Event();

    do {
        Read(*e); // Just hang until user presses a key/button

        e->target->Handle(*e);
    } while (skip==0); // Exit loop on first real keypress


    try_next(*new Tpos(), *trans_array[0], wrapper); // This is it!

// try_next(*new Tpos(), *trans_array[0], route); // This is it!

}

Ivtrans::Ivtrans()
{
    output = new Painter();
    input = new Sensor;
    input->Catch(DownEvent);
    input->Catch(KeyEvent);
}

void Ivtrans::Reconfig()
{
    shape->Rect(width, height);
    shape->Rigid(width / 2, width * 2, height / 2, height * 2);
}


// Redraw is automatically called when the window becomes top-window.

// (l, b) and (r, t) is bounding coordinates of area to be refreshed

void Ivtrans::Redraw(Coord l, Coord b, Coord r, Coord t)
{
    board_print(); // Print out the board

    xdiam = (Coord) xmax/colmax; // Needed when running without Window

    ydiam = (Coord) ymax/rowmax; // Manager. With WM we only need this code

                                 // in Resize()

    xr = xdiam / 2;
    yr = ydiam / 2;
}
```

```
void Ivtrans::Resize()
{
   board_print(); // Print out the board

   xdiam = (Coord) xmax/colmax; // Needed when running without Window

   ydiam = (Coord) ymax/rowmax; // Manager. With a WM we only need this

                                // code in Resize()

   xr = xdiam / 2;
   yr = ydiam / 2;
}

void Ivtrans::Handle(Event &e)
{
   switch (e.eventType) {
   case DownEvent: // User pressed mousebutton

      {
         switch (e.button) {
         case RIGHTMOUSE:
            exit(0);
         case MIDDLEMOUSE:
            Undo* SC = new Undo; // Before routing


            write_mighty_input("mighty.in", *SC);
            if (system("mighty < mighty.in > mighty.out 2> /dev/null")
≠ NULL) {
               cout ≪ "No routing possible.\n";
            }
            else {
               read_mighty_output("mighty.out", *SC); // Read in data

               board_print(); // Show changes

               refresh = TRUE;
            }

            SC→restore(); // Restore situation to before routing

            delete SC;
            break;
         default: skip = 1; break;
         }
         break;
```

```
    }
  case KeyEvent: // User pressed key

    {
      if (e.len > 0) {
        switch (e.keystring[0]) { // Ordinary key

        case 'q':
          exit(0);
        case 'w':
          Undo* SC = new Undo; // Before routing


          write_mighty_input("mighty.in", *SC);
          if (system("mighty < mighty.in > mighty.out 2>
/dev/null") ≠ NULL) {
            cout ≪ "No routing possible.\n";
          }
          else {
            read_mighty_output("mighty.out", *SC); // Read in data

            board_print(); // Show changes

            refresh = TRUE;
          }

          SC→restore(); // Restore situation to before routing

          delete SC;
          break;
        case 'a': skip = 10; break;
        case 's': skip = 100; break;
        case 'd': skip = 1000; break;
        case 'f': skip = 10000; break;
        case 'g': skip = 100000; break;
        case 'h': skip = 1000000; break;
        default: skip = 1; break;
        }
      }
      break;
    }
  default: InternErr();
  }
}
#endif /* IV */
```

placebounds.h

```
/* -*- C++ -*- $Id: $ */

#ifndef _placebounds_h
#define _placebounds_h

#include "main.h"

// ********


class Pos;
class Elem;
class Gpos;
class Tpos;
class Trans;
class Undo;

extern int (*bound[])(Pos&, Elem&, Undo&);

extern void place_trans(Tpos& Tp, Trans& Tr, Undo& SaveChange);
extern void place_diff(Gpos& Gp, Undo &SaveChange);
extern void place_poly(Gpos& Gp, Undo &SaveChange);
extern void place_m1(Gpos& Gp, Undo &SaveChange);
extern void place_m2(Gpos& Gp, Undo &SaveChange);
extern void place_via(Gpos& Gp, Undo &SaveChange);
extern void place_m1_in(Gpos& Gp, int dir, Undo &SaveChange);
extern void place_extra_in(Gpos& gp, int dir, int elem, Undo &SaveChange);

#endif /* _placebounds_h */
```

placebounds.cc

```
/* $Id: main.cc,v 1.17 1991/10/11 14:58:43 dash Exp $ */

#include "main.h"
#include "undo.h"
#include "tpos.h"
#include "trans.h"
#include "terminal.h"
#include "grid.h"
#include "net.h"
#include "vlsi.h"
#include "geometry.h"

#include <stdio.h>

// #include "try_next.h"


// #include "placebounds.h"


// Here comes the routines to be put into the bounds-array. They will

// be called in sequence as long as they return TRUE. Changes in data

// structure must be done via Undo & SaveChange. That way try_next()

// will take care of backtracking.


// Check if it is legal to place the trans on the grid at Tp

int check_bare(Tpos& Tp, Trans& Tr, Undo& SaveChange)
{
  int eli; // Element index


  // Check the point we want to place

  for(eli=0; eli<grid[Tp].nr_elems; eli++)
    if (element_connect[Tp.direction][grid[Tp].elements[eli]][4] == ILLEGAL)
      return FALSE;

  return TRUE;
}

// Check if it is legal to place the trans on the grid at Tp
```

```
int check_diff(Tpos& Tp, Trans& Tr, Undo& SaveChange)
{
   int eli; // Element index


   // This is only allowed if there is already a diff of the right node there

   if (Tp.diff_outside_grid()) // check if diff lands outside grid

      // if (grid(Tp.diff_coord()).elements[WIRE_DIFF]

      // && (grid(Tp.diff_coord()).net_node[WIRE_DIFF] != Tp.diff_node()))

      return FALSE; // @@@ Husk } sjekke mot Transistor diff ogs}


   return TRUE;
}

// Check if it is legal to place the trans on the grid at Tp

int check_illegal(Tpos& Tp, Trans& Tr, Undo& SaveChange)
{
   int eli; // Element index


   // Check neighbors

   for (N8pos nabo(Tp); nabo.legal_pos(); nabo++)
         for(eli=0; eli<grid[Tp].nr_elems; eli++)
            if (element_connect[Tp.direction][grid[Tp].elements[eli]][nabo.nn]
                  == ILLEGAL)
               return FALSE;

   return TRUE;
}

// Check if the connections to the neighbors are OK

int connectivity(Tpos& Tp, Trans& Tr, Undo& SaveChange)
{
   for (N8pos nabo(Tp); nabo.legal_pos(); nabo++) // For each neighboring point

      for(int eli=0; eli<grid[nabo].nr_elems; eli++) { // Go thru each element

         switch(element_connect[Tp.direction][grid[nabo].elements[eli]][nabo.nn]) {
         case (CONN_S):
```

```
        switch(element_connect[grid[nabo].elements[eli]][Tp.direction][8-nabo.nn]) {
        case (CONN_S):
          if (grid[nabo].elements[eli] < 4 )
            if (Tr.source→np ≠ grid[nabo].trns→source→np)
              return FALSE;
          break;
        case (CONN_D):
          if (grid[nabo].elements[eli] < 4 )
            if (Tr.source→np ≠ grid[nabo].trns→drain→np)
              return FALSE;
          break;
        case (CONN_G):
          if (grid[nabo].elements[eli] < 4 )
            if (Tr.source→np ≠ grid[nabo].trns→gate→np)
              return FALSE;
          break;
        case (CONNECTED):
          printf("Error in connectivity, no CONNECTED
implemented\n");
          return FALSE;
          break;
        default: InternErr();
          return FALSE;
          break;
        }
        break;
      case (CONN_D):
        switch(element_connect[grid[nabo].elements[eli]][Tp.direction][8-nabo.nn]) {
        case (CONN_S):
          if (grid[nabo].elements[eli] < 4 )
            if (Tr.drain→np ≠ grid[nabo].trns→source→np)
              return FALSE;
          break;
        case (CONN_D):
          if (grid[nabo].elements[eli] < 4 )
            if (Tr.drain→np ≠ grid[nabo].trns→drain→np)
              return FALSE;
          break;
        case (CONN_G):
          if (grid[nabo].elements[eli] < 4 )
            if (Tr.drain→np ≠ grid[nabo].trns→gate→np)
              return FALSE;
          break;
        case (CONNECTED):
          printf("Error in connectivity, no CONNECTED
implemented\n");
          return FALSE;
          break;
```

```
        default: InternErr();
          return FALSE;
          break;
        }
        break;
      case (CONN_G):
        switch(element_connect[grid[nabo].elements[eli]][Tp.direction][8-nabo.nn]) {
        case (CONN_S):
          if (grid[nabo].elements[eli] < 4 )
            if (Tr.gate→np ≠ grid[nabo].trns→source→np)
              return FALSE;
          break;
        case (CONN_D):
          if (grid[nabo].elements[eli] < 4 )
            if (Tr.gate→np ≠ grid[nabo].trns→drain→np)
              return FALSE;
          break;
        case (CONN_G):
          if (grid[nabo].elements[eli] < 4 )
            if (Tr.gate→np ≠ grid[nabo].trns→gate→np)
              return FALSE;
          break;
        case (CONNECTED):
          printf("Error in connectivity, no CONNECTED
implemented\n");
          return FALSE;
          break;
        default: InternErr();
          return FALSE;
          break;
        }
        break;
      case (CONN_S_DIR):
        switch(element_connect[grid[nabo].elements[eli]][Tp.direction][8-nabo.nn]) {
        case (CONN_S_DIR):
          if (grid[nabo].elements[eli] < 4 )
            if (Tr.source→np ≠ grid[nabo].trns→source→np)
              return FALSE;
            else if (Tr.source→np→term_count ≠ 2)
              return FALSE;
          break;
        case (CONN_D_DIR):
          if (grid[nabo].elements[eli] < 4 )
            if (Tr.source→np ≠ grid[nabo].trns→drain→np)
              return FALSE;
            else if (Tr.source→np→term_count ≠ 2)
              return FALSE;
          break;
```

```
            default: InternErr();
              return FALSE;
              break;
            }
            break;
          case (CONN_D_DIR):
            switch(element_connect[grid[nabo].elements[eli]][Tp.direction][8-nabo.nn]) {
            case (CONN_S_DIR):
              if (grid[nabo].elements[eli] < 4 )
                if (Tr.drain→np ≠ grid[nabo].trns→source→np)
                  return FALSE;
                else if (Tr.source→np→term_count ≠ 2)
                  return FALSE;
              break;
            case (CONN_D_DIR):
              if (grid[nabo].elements[eli] < 4 )
                if (Tr.drain→np ≠ grid[nabo].trns→drain→np)
                  return FALSE;
                else if (Tr.source→np→term_count ≠ 2)
                  return FALSE;
              break;
            default: InternErr();
              return FALSE;
              break;
            }
            break;
          case (CAN_CONN_S):
            printf("Error in connectivity, no CAN_CONN_S
implemented\n");
            return FALSE;
            break;
          case (CAN_CONN_D):
            printf("Error in connectivity, no CAN_CONN_D
implemented\n");
            return FALSE;
            break;
          case (CAN_CONN_G):
            switch(element_connect[grid[nabo].elements[eli]][Tp.direction][8-nabo.nn]) {
            case (CAN_CONNECT):
              break;
            default: InternErr();
              return FALSE;
              break;
            }
            break;
          case (NO_EFFECT):
            break;
          case (ILLEGAL):
```

```
            return FALSE;
            break;
          default:
            printf("Error: type %d, nabo.nn %d, connect %d\n",
                  Tp.direction,
                  nabo.nn,
                  element_connect[Tp.direction][grid[nabo].elements[eli]][nabo.nn]);
            break;
          } // switch

      } // for (int eli ... )


    void place_trans(Tpos& Tp, Trans& Tr, Undo& SaveChange);

    place_trans(Tp, Tr, SaveChange);

    return(TRUE);
}


int liberties(Tpos& Tp, Trans& Tr, Undo& SC)
{
    return (TRUE);
}


// Declare bound as array of pointer to function returning int

// NB: These should by sorted by bounding power for efficiency

// int (*bound[])(Pos &, Elem &, Undo&) = {

int (*bound[])(...) = {
    check_bare,
    check_diff,
    check_illegal,
    connectivity,
    NULL};


void place_trans(Tpos& tp, Trans& tr, Undo& SaveChange)
{
    SaveChange[ (int*) &grid[tp].elements[grid[tp].nr_elems] ];
    grid[tp].elements[grid[tp].nr_elems] = tp.direction;

    SaveChange[ grid[tp].nr_elems ];
    grid[tp].nr_elems = grid[tp].nr_elems+1;
```

```
   tr.geomp = &grid[tp];
   SaveChange[ (int*) &grid[tp].trns ];
   grid[tp].trns = &tr;

   SaveChange[ grid[tp].trans_dir ] = (int)tp.direction;
}


void place_diff(Gpos& gp, Undo &SaveChange)
{
   SaveChange[ grid[gp].elements[grid[gp].nr_elems] ] = WIRE_DIFF;
   SaveChange[ grid[gp].nr_elems ] = grid[gp].nr_elems+1;
}


void place_poly(Gpos& gp, Undo &SaveChange)
{
   SaveChange[ grid[gp].elements[grid[gp].nr_elems] ] = WIRE_POLY;
   SaveChange[ grid[gp].nr_elems ] = grid[gp].nr_elems+1;
}


void place_m1(Gpos& gp, Undo &SaveChange)
{
   SaveChange[ grid[gp].elements[grid[gp].nr_elems] ] = WIRE_M1;
   SaveChange[ grid[gp].nr_elems ] = grid[gp].nr_elems+1;
}

void place_m2(Gpos& gp, Undo &SaveChange)
{
   SaveChange[ grid[gp].elements[grid[gp].nr_elems] ] = WIRE_M2;
   SaveChange[ grid[gp].nr_elems ] = grid[gp].nr_elems+1;
}

void place_via(Gpos& gp, Undo &SaveChange)
{
   SaveChange[ grid[gp].elements[grid[gp].nr_elems] ] = VIA;
   SaveChange[ grid[gp].nr_elems ] = grid[gp].nr_elems+1;
}

void place_extra_in(Gpos& gp, int dir, int elem, Undo &SaveChange)
{
   SaveChange[ grid[gp].elem_dir[dir][elem] ] = TRUE;
}
```

main.cc

```
/* $Id: main.cc,v 1.17 1991/10/11 14:58:43 dash Exp $ */


#include "main.h"
#include "enum.h"
#include "vlsi.h"
#include "undo.h"

int solve_nr = 0; // Number of solutions found

int num_try_next = 0;

#ifdef IV
#include "ivtrans.h"
#include <InterViews/frame.h>
#include <InterViews/world.h>

static PropertyData res[] = {
  {"*font", "*helvetica-medium-r-normal*14*"},
  {nil}
};

static OptionDesc opt[] = {
    {nil}
};

Ivtrans *ivtrans; // Make this visible in ivtrans

#else
#include "try_next.h"
#include "tpos.h"
#include "trans.h"

// Useful to make try_next() call this function whenever it finds a solution

void counter()
{
  solve_nr++;
}

// Useful to make try_next() call this function whenever it finds a solution

void run_mighty()
{
  Undo* SC = new Undo; // Before routing
```

```
  write_mighty_input("mighty.in", *SC);
  if (system("mighty mighty.in mighty.out 2> /dev/null") != 0) {
    cout << "No routing possible.\n";
    system("echo mighty.out");
  } else {
    cout << "Finished routing.\n";
    solve_nr++;
  }
}

#endif /* IV */

// Read a number from commandline (if longer than argnum), or

// ask for a number on stdout, and subsequently read it from stdin.

int read_int_arg(int argnum, char *asktext, int argc, char **argv)
{
  int number;

  if (argnum < argc)
    return (atoi(argv[argnum])); // Read number from commandline...

  do {
    cout << asktext;
    cin.clear();
  } while (!(cin >> number)); // ... or from stdin


  return number;
}

// Read a string from commandline (if longer than argnum), or

// return "input.fil".

char* read_str_arg(int argnum, char *asktext, int argc, char **argv)
{
  if (argnum < argc)
    return ((argv[argnum])); // Read string from commandline...

  return "input.fil";
}


#ifdef PROFILING
// Compile normally.
```

```
// Running prof on resulting mon.out gives total time in each func.


extern "C" int main();
extern "C" void etext();
extern "C" void monstartup(void* first, void* last);
extern "C" void monitor(void* first, void* last = 0, short* buf = 0,
                        int bufsize = 0, int nfuncs = 0);
#endif /* PROFILING */

int main(int argc, char **argv)
{
#ifdef PROFILING
  monstartup(main, etext); // Commence monitoring.

#endif /* PROFILING */

#ifdef IV
  Frame* frame;
  World* world = new World("Base", res, opt, argc, argv);
  char tmp[20];
#endif /* IV */
  char* filename;

  colmax = read_int_arg(1, "Value for colmax?  ", argc, argv);
  assert(0 < colmax && colmax < MAXX);
  rowmax = read_int_arg(2, "Value for rowmax?  ", argc, argv);
  assert(0 < rowmax && rowmax < MAXY);
  transmax = read_int_arg(3, "Value for transmax?  ", argc, argv);
  filename = read_str_arg(4, "Filename for netlist?  ", argc, argv);
  read_netlist(filename);

  grid.init(colmax, rowmax); // Initialize grid


#ifdef IV
  ivtrans = new Ivtrans();
  frame = new BorderFrame(ivtrans);
  sprintf(tmp, "Cell - (%s)", filename);
  frame->SetName(tmp);
  world->Insert(frame);
  ivtrans->Run();
#else
  try_next(*new Tpos(), *trans_array[0], counter); // This is it!

#endif /* IV */

  cout << num_try_next << " placements " << solve_nr << " solutions\n";
```

```
#ifdef PROFILING
  monitor(0); // Stop monitoring and write mon.out.

#endif /* PROFILING */
  return (0);
}
```

geometry.h

```
/* -*- C++ -*- $Id: $ */

#ifndef _geometry_h
#define _geometry_h

#include "main.h"
#include "enum.h"

class Gpos;
class Trans;
class Net;

// ************************************

// Each point in the grid has a Geometry object which describes fully

// all geometrical properties of layout associated with the point

class Geometry {
public:
  Gpos* this_pos; // Point pos. on grid, lower left corner = 0,0

  int trans_dir; // Direction of trans

  int nr_elems; // Nr of elements in this grid-point

  int elements[ELEMENTS]; // Numbers are element-nr. Filled to nr_elems.

  Trans* trns; // Pointer to transistor in x,y, or NULL

  Bool elem_dir[4][ELEMENTS]; // Is element[i] present in direction?

  Net* net_node[ELEMENTS]; /* Nets having wire or contact in x,y */
  int mark[ELEMENTS]; /* Workspace for liberty traversal */

  Geometry();
  Geometry(int, int);
  ~Geometry();
  Geometry& operator=(const Geometry&);
  friend int operator==(const Geometry&, const Geometry&);
  int OK() const; // Checks the representation invariant

};

#endif /* _geometry_h */
```

geometry.cc

```
/* $Id: $ */

#include "main.h"
#include "geometry.h"
#include "gpos.h"

//**************************************


Geometry::Geometry()
{
   this_pos = new Gpos();
   trns = NULL;
   nr_elems = 0;

   for(int tmp=0;tmp<ELEMENTS;tmp++) {
      net_node[tmp] = NULL;
      elements[tmp] = FALSE;
      elem_dir[NORTH][tmp] = FALSE;
      elem_dir[SOUTH][tmp] = FALSE;
      elem_dir[EAST][tmp] = FALSE;
      elem_dir[WEST][tmp] = FALSE;
   }
}

Geometry::Geometry(int xx, int yy)
{
   this_pos = new Gpos(xx, yy);
   trns = NULL;
   nr_elems = 0;

   for(int tmp=0; tmp<ELEMENTS; tmp++) {
      net_node[tmp] = NULL;
      elements[tmp] = FALSE;
      elem_dir[NORTH][tmp] = FALSE;
      elem_dir[SOUTH][tmp] = FALSE;
      elem_dir[EAST][tmp] = FALSE;
      elem_dir[WEST][tmp] = FALSE;
   }
}

Geometry::~Geometry()
{
   delete this_pos;
}
```

```
Geometry& Geometry::operator=(const Geometry& other)
{
   this_pos→x = other.this_pos→x;
   this_pos→y = other.this_pos→y;

   if (trns ≠ other.trns)
      trns = other.trns;

   nr_elems = other.nr_elems;

   for(int tmp=0;tmp<ELEMENTS;tmp++) {
      net_node[tmp] = other.net_node[tmp];
      elements[tmp] = other.elements[tmp];
      elem_dir[NORTH][tmp] = other.elem_dir[NORTH][tmp];
      elem_dir[SOUTH][tmp] = other.elem_dir[SOUTH][tmp];
      elem_dir[EAST][tmp] = other.elem_dir[EAST][tmp];
      elem_dir[WEST][tmp] = other.elem_dir[WEST][tmp];
      mark[tmp] = other.mark[tmp];
   }
   return *this;
}

// Checks the representation invariant

int Geometry::OK() const
{
   return TRUE;
}

int operator==(const Geometry& g1, const Geometry& g2)
{
   assert(g1.OK() && g2.OK());

   if (g1.this_pos→x ≠ g2.this_pos→x
         || g1.this_pos→y ≠ g2.this_pos→y
         || g1.nr_elems ≠ g2.nr_elems)
      return FALSE;
   for(int tmp=0; tmp<g1.nr_elems; tmp++) {
      if (g1.net_node[tmp] ≠ g2.net_node[tmp]
            || g1.mark[tmp] ≠ g2.mark[tmp] // @@@ Trengs denne?

            || g1.elements[tmp] ≠ g2.elements[tmp]
            || g1.elem_dir[NORTH][tmp] ≠ g2.elem_dir[NORTH][tmp]
            || g1.elem_dir[SOUTH][tmp] ≠ g2.elem_dir[SOUTH][tmp]
            || g1.elem_dir[EAST][tmp] ≠ g2.elem_dir[EAST][tmp]
            || g1.elem_dir[WEST][tmp] ≠ g2.elem_dir[WEST][tmp])
         return FALSE;
   }
```

```
    return TRUE;
}
```

grid.h

```
/* -*- C++ -*- $Id: $ */

#ifndef _grid_h
#define _grid_h

#include "main.h"
#include "enum.h"
#include "gpos.h"

class Geometry; // So that we can refer to Geometry


// ************************************


class Grid {
public:
   void init(int, int);
   Geometry& operator[](const Gpos&);
   Geometry& operator()(const Gpos&);
   int OK(); // Checks the representation invariant


// The actual grid, with geometric coordinates in which to place elements

   Geometry* grid_itself[MAXX][MAXY];
};

// Function to access the grid by a Gpos instead of two int's.

inline Geometry& Grid::operator[](const Gpos& gp)
{
   assert(0 ≤ gp.x && gp.x ≤ colmax && 0 ≤ gp.y && gp.y ≤ rowmax);
   return *grid_itself[gp.x][gp.y];
}

#endif /* _grid_h */
```

grid.cc

```
/* $Id: $ */

#include "main.h"
#include "grid.h"
#include "geometry.h"

//**************************************


// set all grid objects to default values

void Grid::init(int colmax, int rowmax)
{
   for(int x=0; x≤colmax; x++)
      for(int y=0; y≤rowmax; y++)
         grid_itself[x][y] = new Geometry(x, y);
}

// Checks the representation invariant

int Grid::OK()
{
   return TRUE;
}
```

net.h

```
/* -*- C++ -*- $Id: $ */

#ifndef _net_h
#define _net_h

#include "main.h"
#include "enum.h"

class Trans;
class Terminal;

// ************************************


// Class describes the nets

class Net {
public:
  char name[NAME_LENGTH]; /* Name of the Net */

  Net(int);

  void include(Terminal*);
  int OK(); // Checks the representation invariant


// private: // Make all public for debugging


  int net_index; /* Array index > 0 < MAX_NET */
  Terminal* nts[MAX_TERMS]; /* pointers to all the Terminals */
  Bool connected; /* whether all Terminals are connected */
  Bool dir_conn; /* TRUE means the Net has only two,
                                        directly connected, Terminals
                                        Implies connected == TRUE */
  Trans* source_conn[SD_CONN]; /* source connections to transistors */
  Trans* drain_conn[SD_CONN]; /* drain connections to transistors */
  Trans* gate_conn[GATE_CONN]; /* gate connections to transistors */
// Io* io_conn[IO_CONN]; /* I/O connections through geometry */

  int source_count; // number of source Terminals included

  int drain_count; // number of drain Terminals included

  int gate_count; // number of gate Terminals included
```

```
    int io_count; // number of io Terminals included

    int term_count; // Total number of of Terminals included

};

#endif /* _net_h */
```

net.cc

```
/* $Id: $ */


#include "main.h"
#include "net.h"

// ************************************


Net::Net(int nr)
{
  DEBOUT(1, "new Net (" << nr << ")");
  net_index = nr;
}

void Net::include(Terminal *t)
{
  DEBOUT(1, "include(Terminal *t). t = " << (int) t);
  nts[term_count++] = t; // @@ Skal jeg ta med source_count osv ogs}?

}

// Checks the representation invariant

int Net::OK()
{
  return TRUE;
}
```

terminal.h

```
/* -*- C++ -*- $Id: $ */

#ifndef _terminal_h
#define _terminal_h

#include "main.h"
#include "gpos.h"

class Trans;
class Net;
// ****************************************


// Class describes the terminals

class Terminal {
public:
   Trans* trans;
   Net *np; /* net which the terminal belongs to */

   Terminal(Net*);
   int OK(); // Checks the representation invariant


private:
   int subnet; /* subnet number = termnumber of one of the
                                  terminals included in the subnet */
   int termnumber; /* termnumber within net (< net->term_count) */
   Gpos pnt; /* point of originating geometry */
   int elem_type; /* element type of originating geometry */
   Bool placed; /* whether placed yet */
};

#endif /* _terminal_h */
```

terminal.cc

```
/* $Id: $ */


#include "main.h"
#include "terminal.h"

//************************************


Terminal::Terminal(Net* n)
{
  DEBOUT(1, "new Terminal()");
  np = n;
}

// Checks the representation invariant

int Terminal::OK()
{
  return TRUE;
}
```