Simulateur de circuits digitaux synchrones

Projet pour le cours « Systèmes digitaux » par Jean Vuillemin

Emmanuel Cornet

École normale supérieure

12 novembre 2003

Table des matières

Introduction

Le but de ce projet est de réaliser un programme informatique simulant le comportement logique de cicruits digitaux synchrones.

Ce programme prendra en entrée la *net-list* du circuit dont on cherche à simuler le fonctionnement et les valeurs de ses entrées pour le nombre N de cycles à simuler. Il retourne les valeurs succesives de toutes les sorties du circuit pour chaque cycle simulé.

I Les choix

I.1 Choix du langage de programmation

Nous avons porté notre choix sur le langage `C`. Les autres langages que nous maîtrisons à l'heure actuelle[1] nous ont semblé moins adaptés pour le projet proposé :

- `Perl` est tout indiqué pour le traitement de fichiers de type texte complexes, mais probablement plus fastidieux à utiliser que le `C` dans notre cas de figure (les fichiers de *net-list* tels que nous avons choisi de les défnir sont de l'ordre de la dizaine, au pire de la centaine de lignes, et chaque ligne ne contient que quelques caractères).
- `Pascal` est, à notre sens, un peu désuet comparé au `C`.
- `OCaml` aurait probablement été un très bon choix mais, notre apprentissage de ce langage ne faisant que débuter, notre maîtrise est encore très loin d'être satisfaisante[2].
- Nous avons finalement choisi d'utiliser le `C` qui nous a semblé naturel, en tant que langage impératif, pour ce type de projet, et qui propose de surcroît une bonne gestion de la lecture de fichiers texte.

Nous avons conscience que la conception d'un langage de *net-list* à part entière (*parser*, *lexer*, etc.) alliée à l'utilisation d'un langage fonctionnel comme `OCaml` résoudrait beaucoup de problèmes que nous nous sommes posés lors d'une programmation « directe » en langage C et produirait probablement un code plus compact et plus élégant. Cependant, notre maîtrise de ces outils étant là encore imparfaite, nous avons préféré choisir une méthode certes plus « pédestre », mais dont nous avons une meilleure connaissance. En outre, le choix de `Caml` ou d'un autre langage fonctionnel étant probablement celui d'une grande majorité d'étudiants, nous avons préféré tenter une approche différente.

Pour résumer, les dimensions et la complexité du projet nous apparaissant comme plutôt raisonnables, l'utilisation d'un jeu complet d'outils de compilation (qui se justifierait si l'on construisait un nouveau langage de programmation, suffisamment complexe) nous paraît un peu surdimensionnée, et reviendrait à sortir l'artillerie lourde pour abattre un moucheron... Nous avons préféré choisir des outils plus modestes, que nous maîtrisons bien, quitte à relever le défi lorsqu'il faut trouver des « astuces » pour contourner les limitations des outils choisis. C'est ce type d'approche que nous avons favorisé, de façon globale.

I.2 Choix d'une structure pour le circuit

Pour représenter les circuits digitaux synchrones, nous avons tout d'abord pensé à des structures en arbre. Mais la valeur d'un nœud étant calculée à partir de celles des nœuds situés en amont, les entrées auraient été les feuilles de l'arbre et la sortie, sa racine. Par conséquent nous aurions été contraints de parcourir l'arbre des feuilles vers la racine (ce qui est inhabituel). En outre, la gestion de sorties multiples n'aurait pas été simple. Enfin, si la plupart des opérateurs logiques utilisés dans les circuits à simuler sont binaires (ce

[1] Nous espérons bien apprendre à maîtriser d'autres langages au cours de nos études d'informatique !

[2] Venant d'une filière physique (licence et maîtrise), nous n'avons pas suivi de formation à ce lanage avant cette année.

qui tendrait à nous diriger vers un arbre binaire), ce n'est pas le cas systématiquement (les registres ne prennent qu'une seule entrée).

Nous avons finalement choisi de nous appuyer sur la possibilité de tri topologique d'une *net-list* pour proposer simplement une structure linéaire, où la valeur de chaque nœud peut être calculée en fonction des valeurs des nœuds qui le précèdent (nous verrons que les registres synchrones contenus dans une boucle font nécessairement exception à cette règle).

En `C`, nous avons implanté cette structure sous la forme d'une *liste chaînée dynamique*.

I.3 Le type « node »

De quel type d'éléments serait composée cette liste chaînée ? Plutôt que de définir de multiples petites structures (une pour les entrées, une pour les noeuds « simples », une pour les registres, etc.), nous avons choisi de définir une structure globale de nœud, comprenant toutes les informations nécessaires. Ce n'est pas optimal du point de vue de l'espace mémoire (certains types de nœuds n'ont que faire d'un champ qui est inadapté pour eux – il ne sera pas utilisé), mais nous avons pris le parti de la simplicité étant donné le degré de complexité relativement faible du jeu d'essai minimal en comparaison avec les capacités de mémoire des machines actuelles[3]. Une stratégie minimale en espace disque aurait probablement été beaucoup plus complexe à mettre en œuvre, et dans ce contexte, nous ne sommes pas convaincu que cela soit indispensable. Pour les mêmes raisons, nous avons préféré favoriser la vitesse d'exécution par rapport à la quantité de mémoire utilisée.

Ainsi, le type « node » que nous avons défini contient :

- le nom du nœud ;
- un bit qui indique si le nœud est une entrée ou non ;
- un bit qui indique si le nœud est une sortie ou non ;
- deux pointeurs vers le premier et le second argument de l'équation logique à partir de laquelle on calcule le nœud ;
- l'opérateur à placer entre les deux arguments (ou `z` dans le cas d'un registre) ;
- la valeur du nœud à un instant donné ;
- un pointeur vers le nœud suivant dans la liste chaînée et un autre vers le nœud précédent ;
- un champ utile pour les entrées, qui précise à quelle ligne du fichier de valeurs d'entrées on trouve l'entrée concernée (ce champ n'est nullement indispensable, mais permet, moyennant un coût en espace relativement faible – un entier pour chaque entrée – de gagner du temps) ;
- deux champs contenant le nom des deux arguments utilisés pour cacluler la valeur du nœud : tant que le tri topologique n'a pas été fait, on retient simplement leur nom avant de pouvoir éditer les liens vers eux directement.

I.4 Les *net-list* sur le disque dur

Parmi les choix que nous devions prendre se trouvait la question suivante : les données d'entrée du programme devaient-elles être transmises au clavier peu à peu, pendant l'exécution du programme, ou bien lues dans un fichier texte ?

Là encore, nous avons fait le choix de la simplicité d'utilisation, même si cela a incontestablement allongé le code source.

Il est en effet bien plus pratique de pouvoir éditer tranquillement une *net-list* et un ensemble de valeurs d'entrées dans un fichier texte, en dehors du programme, de l'enregistrer, puis d'exécuter le programme et de lui indiquer où aller chercher ses données. Une

[3] Tous les tests ont été effectués sur une machine très « modeste » dotée d'un processeur Pentium II à 350 MHz, de 384 Mo de mémoire vive et d'un disque dur de 6 Go.

transmission des données d'entrée au programme pendant son exécution est malpratique, nécessite de saisir à nouveau l'ensemble des informations à chaque exécution, et ne donne pas droit à l'erreur.

L'avantage de pouvoir éditer ses *net-list* séparément devient indiscutable dès lors qu'on travaille avec plusieurs *net-list* en même temps.

II Réalisation

II.1 Les programmes réalisés

Nous avons réalisé deux programmes autonomes : `TriTopo` et `Simulateur` ; nous avons pensé, en effet, qu'il était préférable de découpler les deux processus pour pouvoir trier une *net-list* sans nécessairement simuler le circuit qu'elle représente.

Les différentes fonctions des programmes ont été séparées sur plusieurs fichiers différents, à la fois dans un souci de modularité et pour faciliter l'« épépinage » (*debugging*) ainsi que la réutilisation de ces fonctions dans la suite du projet.

Ainsi, si `TriTopo` est autonome, `Simulateur` est composé de `fAcquerir` (lecture de la *net-list* dans le fichier et stockage en mémoire), `fSimuler` (simulation proprement dite) et `Simulateur` contenant la fonction principale.

II.2 Langage sommaire pour les *net-list*

Nous avons cherché la simplicité à tous les points de vue : chaque ligne contient une information distincte ; l'arité et l'étendue des opérateurs n'étant pas l'objet de confusions, nous n'avons pas inclus de parenthèses ; les équations logiques sont composées sous leur forme intuitive.

La structure d'un fichier de *net-list*, quant à elle, correspond à peu près à une organisation « Entrée – Plat – Dessert » : on trouve d'abord les entrées (une par ligne), puis les équations logiques permettant de calculer chacun des nœuds (une par ligne) et enfin les sorties (une par ligne). Chacun des trois champs se termine par un caractère `*`, seul sur une ligne. Ainsi, la *net-list* de l'additionneur complet pourrait être la suivante :

```
a
b
c
*
t = a ^ b
u = a & b
v = t & c
r = u | v
s = t ^ c
*
s
r
*
```

Le format des fichiers d'entrée est dans le même ordre d'idées : chaque ligne de ce fichier contient simplement le nom de l'entrée concernée suivi d'un espace et de la suite de valeurs binaires que prend l'entrée.

II.3 Registres dans une boucle et tri topologique

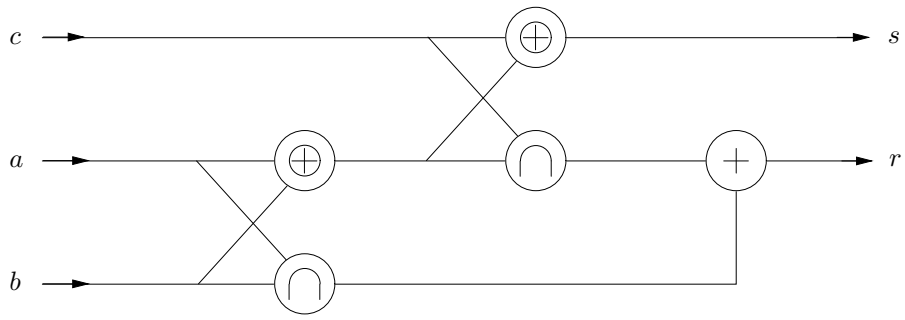
Nous avons fait le choix, pendant le tri topologique, de placer les sorties des registres *avant* les entrées ; il fallait de toutes façons faire un choix puisque les définitions des registres avec boucle ne peut qu'être récursive d'une manière ou d'une autre. Le programme

de simulation du fonctionnement du circuit tient compte de cette convention et n'en accepte pas d'autre (on suppose que le programme travaille sur des *net-list* préalablement triées par `TriTopo`).

Pour les registres à décalage, nous avons respecté la convention de la sortie nulle au temps initial.

III Les circuits simulés

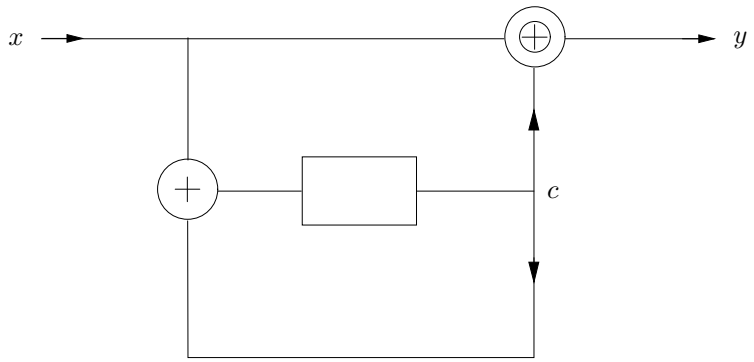
III.1 L'additionneur complet



Ce circuit ne comporte pas de registre (circuit combinatoire) ; il suffit donc de faire sa simulation sur $N = 2^3 = 8$ cycles pour établir sa correction.

a	0	1	0	1	0	1	0	1
b	0	0	1	1	0	0	1	1
c	0	0	0	0	1	1	1	1
s	0	1	1	0	1	0	0	1
r	0	0	0	1	0	1	1	1

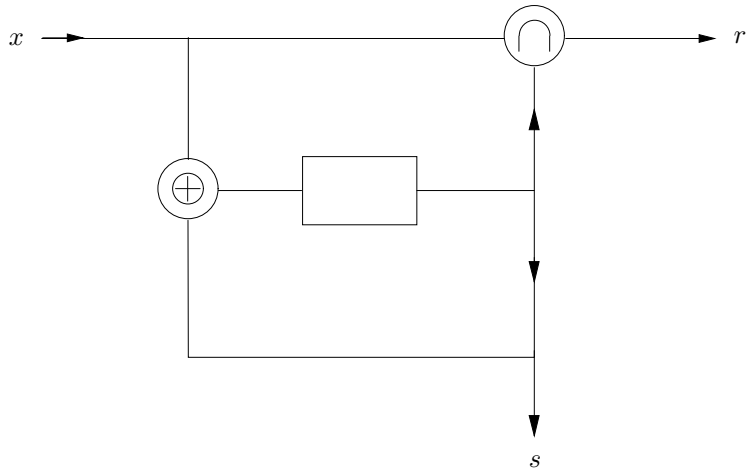
III.2 La fonction $x \mapsto -x$



Ce circuit ne contient qu'un seul registre (et donc deux états) : pour le tester, il suffit de vérifier que le circuit a le comportement attendu pour les deux valeurs d'entrée possibles (0 ou 1) lorsqu'il est dans chacun de ses états. Ici on teste donc sur $2 \times 2 = 4$ cycles (on suppose que le registre donne 0 en sortie à l'instant initial).

x	0	1	1	0
y	0	1	0	1

III.3 Le compteur modulo 2

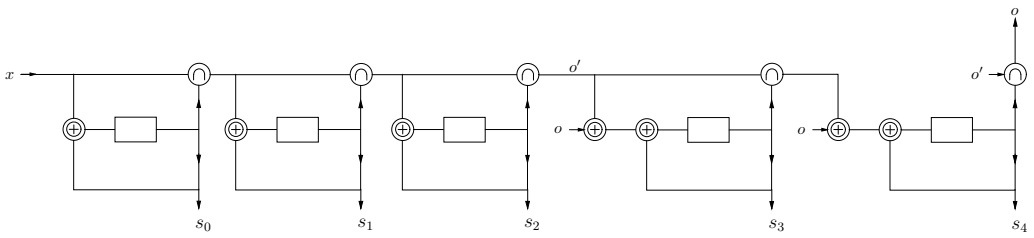


De même que pour $x \mapsto -x$, on teste la succession de valeurs suivante :

x	0	1	1	0
s	0	0	0	1
r	0	0	0	1

III.4 Le compteur modulo 24

Le compteur modulo 24 est simplement composé de trois compteurs modulo 2 auxquels on ajoute on compteur modulo 3.



Il y a cinq registres dans ce circuit, donc 32 états possibles. Le jeu de test complet du circuit devrait donc se faire sur 64 cycles (en alternant, par exemple, les 0 et les 1). Dans notre cas, seuls les 24 premiers états du circuit nous intéressent (on sait qu'il n'atteindra pas les autres états, à condition de le prouver en vérifiant qu'il revient bien à l'état initial après le 24^{e} état). On peut donc tester le circuit sur $(24+1) \times 2 = 50$ cycles.

x	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
s_0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
s_1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0
s_2	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1
s_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
s_4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
o	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

x	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
s_0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
s_1	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
s_2	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0
s_3	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s_4	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
o	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

IV Points forts et limites de la stratégie

IV.1 Ce qui marche bien

La programmation en `C` apporte un certain nombre d'avantages ; en outre, nous nous sommes efforcés de construire notre simulateur de façon à ce qu'il s'adapte aux maximum de situations possible.

- Le code est extrêmement portable. Une simple recompilation suffit à le faire fonctionner sur la plupart des sytèmes informatiques ; tous les systèmes UNIX proposent un compilteur `C`.
- L'exécution du programme `C` est quasi-instantanée. Nous n'avons jamais eu le temps de voir défiler les lignes du calcul avant que le fichier sortie n'ait déjà été écrit. Si cela est évident (et avec n'importe quel langage ou stratégie de programmation) pour le jeu d'essais fourni, les différences avec d'autres langages ou stratégies pourraient peut-être devenir perceptibles sur des données (*net-list* ou nombre de cycles) de taille plus importante.
- Ni la taille de la *net-list*, ni le nombre de cycles ne sont limités en aucune manière. Seule une instruction en « `# define` » limite N à 256, mais cette valeur n'est qu'indicative et ne sert qu'à faciliter la programmation. Il suffit de changer 256 en un nombre plus élevé sur la première ligne du programme.
- Notre choix de placer les sorties des registres situés dans une boucle systématiquement avant leurs entrées est une petite astuce algorithmique qui facilite leur évaluation. En effet, pour un cycle donné, nous pouvons calculer la valeur des sorties de registres directement puisque la valeur des entrées n'a pas encore été mise à jour ; cela évite de stocker les valeurs des entrées des registres du cycle précédent.

IV.2 La contrepartie

Un projet de ce type, réalisé en temps limité, ne peut être qu'imparfait. Même si nous nous sommes efforcés de réaliser nos programmes de manière la moins restrictive possible, il restera toujours des perfectionnements à apporter et des limites à l'approche que nous avons choisie. Tentons d'en identifier certaines :

- L'absence de programmes du type *parser*, *lexer*, etc., ne pose pas les bases explicites du langage simple de *net-list* que nous avons proposé et ne garantit donc pas de réel « blindage » au niveau de la syntaxe des fichiers d'entrée. On suppose donc que le fichier donné au pogramme a, du moins approximativement car beaucoup d'erreurs de syntaxe conduisent tout de même au rejet des données, la structure attendue.

- Nous sommes partis de l'idée que tout circuit digital synchrone pouvait être construit à partir d'une base d'éléments très limitée (de la même manière que, par exemple, on peut reconstruire l'opérateur logique « ou » avec des « et » et des « non » grâce aux lois de De Morgan). L'ajout d'un élément de base demande un certain nombre de modifications dans le code.
- Notre tri topologique est imparfait du côté des registres contenus dans une boucle. Nous avons fait le choix de n'éditer les liens pointant vers les arguments des registres qu'une fois tous les autres liens édités, ce qui autorise *a priori* n'importe quel position dans la *net-list* pour les sorties des registres. Cependant, le tri topologique s'effectuant sur tous les autres nœuds, et en partiulier sur les entrées des registres, les sorties figurent toujours avant les entrées (ce qui est d'ailleurs pratique, comme nous l'avons mentionné). Notons tout de même que cette convention est imposée par notre choix d'un ordre strict dans le tri topologique (deux nœuds ne peuvent pas avoir la même profondeur) imposé par la structure de liste chaînée.

V Code source des programmes réalisés

Tous ces fichiers sont disponible à l'adresse

`www.eleves.ens.fr/home/cornet/Simulateur/`

Tous les programmes sont largement commentés pour faciliter la lecture.

V.1 Librairie `simu.h`

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

#define MAX_CYCLES 256

////////////////////////////////////////////////////////////////////
////////////////////////// Définition d'un noeud ////////////////////////
////////////////////////////////////////////////////////////////////

typedef struct node
{
char * nom;             // Nom du noeud.
int in;                 // Est-ce une entrée ?
int out;                // Est-ce une sortie ?
struct node * arg1;     // Premier noeud à partir duquel on le calcule.
struct node * arg2;     // Second noeud à partir duquel on le calcule.
char op;                // Opérateur à mettre en arg1 et arg2.
int val;                // Valeur (tension) du noeud à un moment donné.
struct node * nxt;      // Noeud suivant dans la netlist ordonnée.
struct node * prv;      // Noeud précédent dans la netlist ordonnée.
int info;               // Pour les entrées : à quelle ligne du
                        // fichier d'entrées on trouve ses valeurs.
char * temp1;           // Le nom des arguments en attendant de
char * temp2;           //  savoir où ils sont.
} node;

////////////////////////////////////////////////////////////////////
////////////////////////// Fonctions de traitement //////////////////////
////////////////////////////////////////////////////////////////////

node * acquerir (node * N);
node * tri_topo (node * N);
node * simuler (node * N);
```

V.2 Programme `Simulateur.c`

```
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
//////// Projet "Simulateur de circuit digital synchrone" du cours /////
//////// "Systèmes digitaux" de Jean Vuillemin. ////////////////////////
//////// ENS 2003/2004. ////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
```

```
//                                          © Emmanuel Cornet, novembre 2003

#include "simu.h"

int main()
{

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////// Variables ///////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

node * N;                 // C'est la net-list sur laquelle on travaille.
node * cur, * next;        // Pour se décplacer dans la liste des noeuds.
char * nom_fich_netlist;        // Le nom de fichier fourni par l'utilis.
char * read;                         // La chaîne courante dans le fichier lu.

nom_fich_netlist = (char*)malloc(256*sizeof(char));
read=(char*)malloc(80*sizeof(char));

// On prend la limite UNIX pour les noms de fichiers (256 caract.).
// et on suppose que les lignes ne seront pas plus longues que 80.


////////////////////////////////////////////////////////////////////////
////////////////////////////////// Accueil /////////////////////////////
////////////////////////////////////////////////////////////////////////

printf( "Bonjour ! Je suis le programme 'Simulateur' ; je sais simuler le
        fonctionnement d'un circuit digital synchrone à partir d'une
        net-list que vous me fournirez et des entrées du circuit sur un
        nombre de cycles donné.\n\n");


////////////////////////////////////////////////////////////////////////
////////////////////////////// Acquisition /////////////////////////////
////////////////////////////////////////////////////////////////////////

N = acquerir (N);

printf("\nJe vais passer à la simulation.\n\n");

////////////////////////////////////////////////////////////////////////
////////////////////////////// Simulation //////////////////////////////
////////////////////////////////////////////////////////////////////////

N = simuler(N);

////////////////////////////////////////////////////////////////////////
////////////////////////////// Nettoyage ///////////////////////////////
////////////////////////////////////////////////////////////////////////


cur = N;
```

```
while (cur != NULL)
        {
        next = (*cur).nxt;
        free(cur);
        cur = next;
        }

printf("\n\nVoilà, j'ai terminé et il semble que tout se soit bien passé.
        Vous trouverez les valeurs de sortie dans le fichier sortie
        \nÀ bientôt !\n");
return 0;
}
```

V.3 Fonction `fAcquerir.c`

```
#include "simu.h"

#define MAX_CYCLES 256

////////////////////////////////////////////////////////////////////////
////////////////////////////// Acquisition /////////////////////////////
////////////////////////////////////////////////////////////////////////

node * acquerir (node * N)
{

////////////////////////////////////////////////////////////////////////
////////////////////////////// Variables ///////////////////////////////
////////////////////////////////////////////////////////////////////////

node * cur, * next;               // Pour se déplacer dans la liste des
                                  // noeuds.
char * nom_fich_netlist;          // Le nom de fichier fourni par l'utilis.
char * read;                        // La chaîne courante dans le fichier lu.
node * jumper;                       // Va servir à parcourir la liste

nom_fich_netlist = (char*)malloc(256*sizeof(char));
read=(char*)malloc(80*sizeof(char));

// On prend la limite UNIX pour les noms de fichiers (256 caract.).
// et on suppose que les lignes ne seront pas plus longues que 80.



////////////////////////////////////////////////////////////////////////
////////////////////////////// Nom du fichier //////////////////////////
////////////////////////////////////////////////////////////////////////

printf("Veuillez tout d'abord m'indiquer le fichier dans lequel se trouve
        la net-list à analyser.\n\n");

scanf("%s", nom_fich_netlist);

FILE * flux_netlist;
if((flux_netlist = fopen (nom_fich_netlist, "r"))==NULL)
```

```
        {
        printf("Erreur de lecture du fichier.\n");
        return NULL;
        }

////////////////////////////////////////////////////////////////////////
///////////////////////////// Initialisation ///////////////////////////
////////////////////////////////////////////////////////////////////////

N=(node *)malloc(sizeof(node));
if (N == NULL) printf("Erreur malloc.\n");
(*N).prv = NULL;
(*N).nxt = NULL;

fscanf(flux_netlist, "%s",read);
if(strcmp(read,"*") == 0)
        {
        printf("Désolé, mais je ne gère pas les circuits sans aucune
                entrée.\n");
        return NULL;
        }
if (strcmp(read, "z")==0)
        {
        printf("Désolé, mais le caractère z est réservé aux
                registres.\n");
        return NULL;
        }
(*N).nom = read;
(*N).in = 1;
(*N).out = 0;

cur = N;

// On initialise le noeud courant au début de la netlist.
// Pour l'instant, N est tout seul donc son suivant et son précédent
// pointent vers NULL.

// On va faire trois boucles différentes : une pour les entrées, une
// pour les sorties et une pour les équations logiques.
// Le délimiteur choisi entre ces trois catégories est "*".

////////////////////////////////////////////////////////////////////////
////////////////////////////// Entrées /////////////////////////////////
////////////////////////////////////////////////////////////////////////

while (strcmp(read,"*") !=0)
        {
        // Création du nouvel espace mémoire.
        next=(node *)malloc(sizeof(node));
        if (next == NULL) printf("Erreur malloc.\n");
        read=(char*)malloc(80*sizeof(char));

        // Il faut réallouer read, sinon on perd (*cur).nom

        fscanf(flux_netlist, "%s", read);
```

```
        if (strcmp(read,"*")==0) break;
        // Édition des liens.

        (*cur).nxt = next;
        (*next).nxt = NULL;
        (*next).prv = cur;
        cur=next;

        // Mise en place des valeurs.

        if (strcmp(read, "z")==0)
                {
                printf("Désolé, mais le caractère z est réservé aux
                        registres.\n");
                return NULL;
                }
        (*cur).nom = read;
        (*cur).in = 1;
        (*cur).out = 0;
        }
read=(char*)malloc(80*sizeof(char));
fscanf(flux_netlist, "%s",read);

////////////////////////////////////////////////////////////////////////
////////////////////////// Équations logiques //////////////////////////
////////////////////////////////////////////////////////////////////////

while (strcmp(read,"*") !=0)
        {

        // Création du nouvel espace mémoire.

        next=(node *)malloc(sizeof(node));
        if (next == NULL) printf("Erreur malloc.\n");

        // Édition des liens

        (*cur).nxt = next;
        (*next).nxt = NULL;
        (*next).prv = cur;
        cur=next;

        // Mise en place des valeurs.

        // Nom

        (*cur).nom = read;
        read=(char*)malloc(80*sizeof(char));
        fscanf(flux_netlist, "%s",read);

////////////////////////////// Signe "égal" ////////////////////////////

        if(strcmp(read,"=") != 0)
                {
```

```
                printf("Erreur de syntaxe dans les équations logiques");
                }
        read=(char*)malloc(80*sizeof(char));
        fscanf(flux_netlist, "%s", read);
        jumper = N;

//////////////////////////// Premier argument ////////////////////////////

        (*cur).arg1 = NULL;

/////// On traite le cas particulier des registres

        if(strcmp(read,"z")==0)
                {
                (*cur).op = 'z';
                read=(char*)malloc(80*sizeof(char));
                fscanf(flux_netlist, "%s", read);

                // On s'avance d'un cran pour aller prendre l'argument et
                // on s'arrête là.

                (*cur).temp1 = read;

// On remet la tête de lecture là où il faut pour les suivants.

                read=(char*)malloc(80*sizeof(char));
                fscanf(flux_netlist, "%s", read);

                continue;
                }

        // On va rechercher ce premier argument dans la liste.

        while(jumper != NULL)
                {
                if (strcmp((*jumper).nom, read) == 0)
                        {
                        (*cur).arg1 = jumper;
                        break;}
                jumper = (*jumper).nxt;
                }

        // Si le pointeur est toujours nul, c'est qu'il y a un problème

        if ((*cur).arg1 == NULL)
                {
                printf("Le tri topologique de la net-list est
                        nécessaire.\n");
                return NULL;
                }

////////////////////////////// Opérateur ////////////////////////////

        (*cur).op = fgetc(flux_netlist);
        (*cur).op = fgetc(flux_netlist);
```

```
        // On se déplace du nombre de caractères qu'il faut pour
        // lire l'op.

        read = (char *)fgetc(flux_netlist);
        read=(char*)malloc(80*sizeof(char));
        fscanf(flux_netlist, "%s", read);

/////////////////////////// Deuxième argument ////////////////////////

        (*cur).arg2 = NULL;

        // On va rechercher ce premier argument dans la liste.

        jumper = N;
        while(jumper != NULL)
                {
                if (strcmp((*jumper).nom, read) == 0)
                        {(*cur).arg2 = jumper;break;}
                jumper = (*jumper).nxt;
                }

        // Si le pointeur est toujours nul, c'est qu'il y a un problème

        if ((*cur).arg2 == NULL)
                {
                printf("Le tri topologique de la net-list est
                        nécessaire.\n");
                return NULL;
                }
        read=(char*)malloc(80*sizeof(char));
        fscanf(flux_netlist, "%s", read);
        }

read=(char*)malloc(80*sizeof(char));
fscanf(flux_netlist, "%s",read);
if(strcmp(read,"*")==0)
        printf("Désolé, mais il me faut au moins une sortie.\n");

////////////////////////////////////////////////////////////////////////
////////////////////// Retour sur les registres ////////////////////////
////////////////////////////////////////////////////////////////////////

// Maintenant que tous les noeuds ont été définis, il faut faire
// l'édition des liens vers les arguments des registres.

cur = N;
while (cur != NULL)
        {
        if ((*cur).op == 'z')
                {
                jumper = N;
                while(jumper != NULL)
                        {
                        if (strcmp((*jumper).nom, (*cur).temp1) == 0)
```

```
                                {
                                (*cur).arg1 = jumper;
                                (*cur).arg2 = NULL;
                                break;}
                        jumper = (*jumper).nxt;
                        }

                // Si le pointeur est toujours nul, c'est qu'il y a
                // un problème

                if ((*cur).arg1 == NULL)
                        printf("Il y a eu une erreur sur l'argument
                                des registres. L'un d'eux n'est peut-être
                                pas défini.\n");
                }
        cur=(*cur).nxt;
        }
////////////////////////////////////////////////////////////////////////
////////////////////////////////// Sorties /////////////////////////////
////////////////////////////////////////////////////////////////////////

// Maintenant on ne crée plus d'espace mémoire, on se contente de mettre
// à 1 le bit "out" des noeuds qui sont des sorties.

while (strcmp(read,"*") !=0)
        {
        jumper = N;
        while(jumper != NULL)
                {
                if (strcmp((*jumper).nom, read) == 0)
                          {
                        (*jumper).out = 1;
                        break;
                        }
                jumper = (*jumper).nxt;
                }
        read=(char*)malloc(80*sizeof(char));
        fscanf(flux_netlist, "%s",read);
        }
fclose(flux_netlist);

return N;
}
```

V.4 Fonction `fSimuler.c`

```
#include "simu.h"

////////////////////////////////////////////////////////////////////////
////////////////////////////// Simulation //////////////////////////////
////////////////////////////////////////////////////////////////////////

node * simuler(node * N)
{
```

```
////////////////////////////////////////////////////////////////////////
////////////////////////////// Variables ///////////////////////////////
////////////////////////////////////////////////////////////////////////

node * cur;                        // Pour se décplacer dans la liste des
                                // noeuds.
char * read;                         // La chaîne courante dans le fichier lu.
int M;                               // Nombre de cycles à simuler.
char * nom_fich_entrees;        // Fichier contenant les entrées.
int i;                               // Un compteur.
                                // chaînée à la recherche d'un élément.
int t;                               // Compteur de temps (cycles).
char bit;                         // Lit les bits d'entrée.
int a,b;                        // Les deux arguments pour calculer la
                                // valeur d'un d'noeud.

nom_fich_entrees = (char*)malloc(256*sizeof(char));
read=(char*)malloc(80*sizeof(char));

// On prend la limite UNIX pour les noms de fichiers (256 caract.).
// et on suppose que les lignes ne seront pas plus longues que 80.


////////////////////////////////////////////////////////////////////////
//////////////////////////// Nombre de cycles ? ////////////////////////
////////////////////////////////////////////////////////////////////////

printf("Veuillez maintenant m'indiquer le nombre de cycles que je dois
        simuler ; attention : vous devez avoir fourni suffisamment
        d'informations dans le fichier contenant les valeurs
        d'entrée !\n\n");

scanf("%i",&M);
if (M < 1)
        {
        printf("Désolé, mais je ne peux pas simuler un nombre de
        cycles négatif ou nul.\n");
        return NULL;
        }

////////////////////////////////////////////////////////////////////////
//////////////////////////// Fichier d'entrées /////////////////////////
////////////////////////////////////////////////////////////////////////


printf("\nMerci. Maintenant j'ai besoin que vous me donniez le nom du
        fichier contenant les valeurs d'entrée.\n\n");


scanf("%s", nom_fich_entrees);

FILE * flux_entrees;
if((flux_entrees = fopen (nom_fich_entrees, "r"))==NULL)
        {
```

```
        printf("Erreur de lecture du fichier.\n");
        return NULL;
        }
printf("\n");

// On va commencer par remplir les cases "info" des entrées, à savoir
// regarder à quelle position du fichier d'entrées chacune d'elle se
// trouve (pour ne pas avoir à la rechercher à chaque fois).

cur = N;
while(cur != NULL)
        {
        if ((*cur).in == 0) break;

        // Les entrées sont toujours au début de la liste chaînée.

        rewind(flux_entrees); i=0;
        while(fgets(read, MAX_CYCLES, flux_entrees)!=NULL)
                {
                if (strncmp((*cur).nom,read,strlen((*cur).nom))==0)
                        {(*cur).info = i; break;}
                i++;
                }
        cur = (*cur).nxt;
        }

//cur = N;
//while (cur != NULL)
//        {
//        if ((*cur).in == 1)
//        cur = (*cur).nxt;
//        }

////////////////////////////////////////////////////////////////////////
////////////////////// La vraie boucle de simulation ///////////////////
////////////////////////////////////////////////////////////////////////

FILE * sortie;
sortie = fopen("sortie", "w");
for (t=0 ; t<M ; t++)
        {
        printf(".");
        cur = N;
        fprintf(sortie,"\nCycle numéro %i\n\n",t);
        while (cur != NULL)
                {
////////////////// Ce qu'il faut faire pour les entrées /////////////////

                if ((*cur).in == 1)
                        {
                        rewind(flux_entrees);

                        // On se déplace du nbre de lignes nécessaire

                        for (i=0 ; i < (*cur).info ; i++)
```

```
                                {
                                fgets(read,80,flux_entrees);
                                }

                        // On se déplace du nbre de colonnes nécessaire

                        fscanf(flux_entrees,"%s",read);
                        bit = fgetc(flux_entrees);

                        // Pour sauter le nom de l'entrée et l'espace
                        // d'après

                        for (i=0 ; i < t + 1 ; i++)
                                {
                                bit = fgetc(flux_entrees);
                                }
                        switch (bit)
                                {
                                case '0' : (*cur).val = 0; break;
                                case '1' : (*cur).val = 1; break;

                                // On se souvient de leur valeur au
                                // cycle précédent, pour la présence
                                // éventuelle de registres.

                                default : printf("Erreur dans les données
                                                d'entrée.\n");
                                        return NULL; break;
                                }
                        }

////////////////// Ce qu'il faut faire pour les noeuds /////////////////

                if ((*cur).in != 1)
                        {
                        a = (*((*cur).arg1)).val;
                        if ((*cur).op != 'z')
                        b = (*((*cur).arg2)).val;

// A ne pas faire s'il s'agit d'un registre ! (Arg2 pas défini)

                        switch((*cur).op)
                                {
                                case '&' :
                                        (*cur).val = a * b;
                                        break;
                                case '|' :
                                        (*cur).val = a + b - a * b;
                                        break;
                                case '^' :
                                        (*cur).val = (a - b)*(a - b);
                                        break;
                                case 'z' :
                                        if (t == 0)
                                                {
```

```
                                        (*cur).val = 0;
                                        }
                                else
                                        {
                                        (*cur).val =
                                        (*((*cur).arg1)).val;

                                        // Les sorties des
                                        // registres
                                        // sont placées avant les
                                        // entrées (valeur du
                                        // cycle actuel pas encore
                                        // calculées).
                                        }
                                break;
                        default : printf ("Erreur d'opérateur.
                                        Je ne connais pas
                                        l'opérateur %c.\n",
                                        (*cur).op);
                        }
                }

////////////////// Ce qu'il faut faire pour les sorties /////////////////

                // Plus rien à calculer, il suffit d'écrir leur valeur
                // dans le fichier de sortie.

                if ((*cur).out == 1)
                        {
                        fprintf(sortie,"%s = %i\n",(*cur).nom,
                                (*cur).val);
                        }
                cur = (*cur).nxt;
                }
        }
fclose (flux_entrees);
return 0;
}
```

V.5 Programme `TriTopo.c`

```
#include "simu.h"

#define MAX_CYCLES 256

////////////////////////////////////////////////////////////////////////
////////////////////////////// Lecture /////////////////////////////////
////////////////////////////////////////////////////////////////////////

int main ()
{

////////////////////////////////////////////////////////////////////////
////////////////////////////// Variables ///////////////////////////////
////////////////////////////////////////////////////////////////////////
```

```
node * N;                          // La net-liste
node * cur, * next;                  // Pour se déplacer dans la liste des
                                   // noeuds.
char * nom_fich_netlist;           // Le nom de fichier fourni par l'utilis.
char * read;                            // La chaîne courante dans le fichier lu.
node * jumper;                           // Va servir à parcourir la liste
node * swap;                           // Pour faire une échange entre deux
                                   // éléments.
int test1, test2;                   // Teste si les args sont définis

nom_fich_netlist = (char*)malloc(256*sizeof(char));
read=(char*)malloc(80*sizeof(char));

// On prend la limite UNIX pour les noms de fichiers (256 caract.).
// et on suppose que les lignes ne seront pas plus longues que 80.



////////////////////////////////////////////////////////////////////////
///////////////////////////// Nom du fichier ///////////////////////////
////////////////////////////////////////////////////////////////////////

printf("Veuillez tout d'abord m'indiquer le fichier dans lequel se
        trouve la net-list à analyser.\n\n");

scanf("%s", nom_fich_netlist);

FILE * flux_netlist;
if((flux_netlist = fopen (nom_fich_netlist, "r"))==NULL)
        {
        printf("Erreur de lecture du fichier.\n");
        return -1;
        }

////////////////////////////////////////////////////////////////////////
///////////////////////////// Initialisation ///////////////////////////
////////////////////////////////////////////////////////////////////////

N=(node *)malloc(sizeof(node));
if (N == NULL) printf("Erreur malloc.\n");
(*N).prv = NULL;
(*N).nxt = NULL;

fscanf(flux_netlist, "%s",read);
if(strcmp(read,"*") == 0)
        {
        printf("Désolé, mais je ne gère pas les circuits sans aucune
                entrée.\n");
        return -1;
        }
if (strcmp(read, "z")==0)
        {
        printf("Désolé, mais le caractère z est réservé aux
                registres.\n");
```

```
        return -1;
        }
(*N).nom = read;
(*N).in = 1;

cur = N;

// On initialise le noeud courant au début de la netlist.
// Pour l'instant, N est tout seul donc son suivant et son précédent
// pointent vers NULL.

// On va faire trois boucles différentes : une pour les entrées, une
// pour les sorties et une pour les équations logiques.
// Le délimiteur choisi entre ces trois catégories est "*".

////////////////////////////////////////////////////////////////////////
////////////////////////////// Entrées /////////////////////////////////
////////////////////////////////////////////////////////////////////////

while (strcmp(read,"*") !=0)
        {
        // Création du nouvel espace mémoire.
        next=(node *)malloc(sizeof(node));
        if (next == NULL) printf("Erreur malloc.\n");
        read=(char*)malloc(80*sizeof(char));

        // Il faut réallouer read, sinon on perd (*cur).nom

        fscanf(flux_netlist, "%s", read);

        if (strcmp(read,"*")==0) break;
        // Édition des liens.

        (*cur).nxt = next;
        (*next).nxt = NULL;
        (*next).prv = cur;
        cur=next;

        // Mise en place des valeurs.

        if (strcmp(read, "z")==0)
                {
                printf("Désolé, mais le caractère z est réservé aux
                        registres.\n");
                return -1;
                }
        (*cur).nom = read;
        (*cur).in = 1;
        }
read=(char*)malloc(80*sizeof(char));
fscanf(flux_netlist, "%s",read);

////////////////////////////////////////////////////////////////////////
////////////////////////// Équations logiques ///////////////////////////
////////////////////////////////////////////////////////////////////////
```

```
while (strcmp(read,"*") !=0)
        {

        // Création du nouvel espace mémoire.

        next=(node *)malloc(sizeof(node));
        if (next == NULL) printf("Erreur malloc.\n");

        // Édition des liens

        (*cur).nxt = next;
        (*next).nxt = NULL;
        (*next).prv = cur;
        cur=next;

        // Mise en place des valeurs.

        // Nom

        (*cur).nom = read;
        read=(char*)malloc(80*sizeof(char));
        fscanf(flux_netlist, "%s",read);

////////////////////////////// Signe "égal" ////////////////////////////

        if(strcmp(read,"=") != 0)
                {
                printf("Erreur de syntaxe dans les équations logiques");
                }
        read=(char*)malloc(80*sizeof(char));
        fscanf(flux_netlist, "%s", read);
        jumper = N;

////////////////////////// Premier argument ////////////////////////////

/////// On traite le cas particulier des registres

        if(strcmp(read,"z")==0)
                {
                (*cur).op = 'z';
                read=(char*)malloc(80*sizeof(char));
                fscanf(flux_netlist, "%s", read);

                // On s'avance d'un cran pour aller prendre l'argument
                // et on s'arrête là.

                (*cur).temp1 = read;

// On remet la tête de lecture là où il faut pour les suivants.

                read=(char*)malloc(80*sizeof(char));
                fscanf(flux_netlist, "%s", read);

                continue;
```

```
                }
        (*cur).temp1 = read;

////////////////////////////// Opérateur ////////////////////////////

        (*cur).op = fgetc(flux_netlist);
        (*cur).op = fgetc(flux_netlist);

        // On se déplace du nombre de caractères qu'il faut pour lire
        // l'op.

        read = (char *)fgetc(flux_netlist);
        read=(char*)malloc(80*sizeof(char));
        fscanf(flux_netlist, "%s", read);

////////////////////////// Deuxième argument ////////////////////////

        (*cur).temp2 = read;

        read=(char*)malloc(80*sizeof(char));
        fscanf(flux_netlist, "%s", read);
        }

read=(char*)malloc(80*sizeof(char));
fscanf(flux_netlist, "%s",read);
if(strcmp(read,"*")==0) printf("Désolé, mais il me faut au moins
                                une sortie.\n");

////////////////////////////////////////////////////////////////////////
////////////////////////////// Sorties /////////////////////////////////
////////////////////////////////////////////////////////////////////////

while (strcmp(read,"*") != 0)
        {
        jumper = N;
        while (jumper != NULL)
                {
                if (strcmp((*jumper).nom,read) == 0)
                        {
                        (*jumper).out = 1;
                        break;
                        }
                jumper = (*jumper).nxt;
                }
        read = (char *) malloc (80 * sizeof(char));
        fscanf(flux_netlist, "%s", read);
        }

fclose(flux_netlist);

////////////////////////////////////////////////////////////////////////
////////////////////////// Tri topologique /////////////////////////////
////////////////////////////////////////////////////////////////////////

cur = N;
```

```
// On n'a pas besoin de vérifier les entrées ; on se déplace donc
// jusqu'au premier noeud qui n'est pas une entrée.

while ((*cur).in == 1) cur = (*cur).nxt;

while(cur != NULL)
        {
        test1 = 0; test2 = 0;
        jumper = N;
// Cas particulier : registre ; on ne fait rien du tout puisque
// l'édition des liens vers le paramètre se fait après celle de tous
// les autres noeuds

        if ((*cur).op == 'z')
                {
                cur = (*cur).nxt;
                continue;
                }
        else
                {
                while (jumper != cur)
                        {
                        if (strcmp((*cur).temp1,(*jumper).nom) == 0)
                                test1 = 1;
                        if (strcmp((*cur).temp2,(*jumper).nom) == 0)
                                test2 = 1;
                        if (test1 * test2 == 1) break;
                        jumper = (*jumper).nxt;
                        }
                if (test1 * test2 == 1)
                        {
                        cur = (*cur).nxt;
                        continue;
                        }
                }
        jumper = cur;
        while ((*jumper).nxt != NULL)
                {
                jumper = (*jumper).nxt;
                }

        // Maintenant si l'on est encore dans la boucle, c'est que les
        // arguments du noeud ne sont pas définis. On va mettre le
        // noeud à la fin de la liste.

        // On commence par se souvenir du nxt de cur pour savoir où
        // continuer à la prochaine boucle

        swap = (*cur).nxt;

        // 1er cas : c'est le premier de la liste

        if (cur == N)
                {
```

```
                (*((*cur).nxt)).prv = NULL;
                (*cur).nxt = NULL;
                (*cur).prv = jumper;
                (*jumper).nxt = cur;
                }

        // 2e cas : c'est le dernier ; ça veut dire que la net-liste
        // est mauvaise

        if ((*cur).nxt == NULL)
                {
                printf("Il y a un problème dans votre net-list.
                        Je n'arrive pas à évaluer l'un des arguments du
                        noeud %s\n",(*cur).nom);
                return -1;
                }
        else
                {
                (*((*cur).prv)).nxt = (*cur).nxt;
                (*((*cur).nxt)).prv = (*cur).prv;
                (*cur).nxt = NULL;
                (*cur).prv = jumper;
                (*jumper).nxt = cur;
                }

        cur = swap;
        }

////////////////////////////////////////////////////////////////////////
////////////////////// Écriture du résultat ////////////////////////////
////////////////////////////////////////////////////////////////////////

FILE * nltriee;
nltriee = fopen ("net-list_triee", "w");

cur = N;

////////////////////// Écriture des entrées ////////////////////////////

while ((*cur).in == 1)
        {
        fprintf(nltriee, "%s\n", (*cur).nom);
        cur = (*cur).nxt;
        }

fprintf(nltriee,"*\n");

////////////////////// Écriture des équations //////////////////////////

while (cur != NULL)
        {
        if ((*cur).op == 'z')
                {
                fprintf(nltriee, "%s = z %s\n",(*cur).nom, (*cur).temp1);
                }
```

```
        else
                {
                fprintf(nltriee, "%s = %s %c %s\n",(*cur).nom,
                        (*cur).temp1,(*cur).op, (*cur).temp2);
                }
        cur = (*cur).nxt;
        }

fprintf(nltriee,"*\n");

////////////////////// Écriture des sorties ////////////////////////

cur = N;
while (cur != NULL)
        {
        if ((*cur).out == 1)
                {
                fprintf(nltriee,"%s\n", (*cur).nom);
                }
        cur = (*cur).nxt;
        }

fprintf(nltriee,"*\n");

fclose (nltriee);
return 0;
}
```