THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

SPÉCIALITÉ : INFORMATIQUE

Présentée par Frédéric PÉTROT
pour obtenir le titre de
Docteur de l'Université Paris VI

OUTILS D'AIDE AU DÉVELOPPEMENT DE BIBLIOTHÈQUES VLSI PORTABLES

Soutenue le 8 juillet 1994 devant le jury composé de

Michel WEINFELD	Président
Jean-Claude DUFOURD	Rapporteur
Jean-Pierre SCHOELLKOPF	Rapporteur
Bernard COURTOIS	Examinateur
Alain GREINER	Examinateur
Michel ROBERT	Examinateur
Jean-Pierre TUAL	Examinateur

À mes parents

Remerciements

Ce travail a été effectué sous la direction d'Alain Greiner, qui en est également l'initiateur. Je lui dois beaucoup, entre autre pour les nombreuses et instructives discussions sans lesquels le développement et l'achèvement de ces recherches — et de ce manuscrit — n'auraient été possible.

Que les membres du jury trouvent ici l'expression de ma gratitude. Les rapporteurs tout d'abord : Jean-Claude Dufourd de Télécom Paris pour m'avoir permis d'expérimenter **préforme** et pour ses qualités humaines et scientifiques et Jean-Pierre Schoellkopf de SGS-Thomson pour m'avoir poussé à remettre l'ouvrage sur le métier pour le peaufiner. Le président : Michel Weinfeld, directeur du LIX ; et les examinateurs : Bernard Courtois, responsable du CMP et directeur de TIM3, Michel Robert, du LIRMM, et Jean-Pierre Tual de Bull Systèmes, pour l'intérêt qu'ils ont porté à mes recherches.

Une thèse est le travail d'un individu, mais sa concrétisation est aussi un travail d'équipe, en particulier dans les domaines appliqués comme celui de la CAO pour *VLSI*. Que mes collègues chercheurs qui, en développant les applications de la chaîne Alliance, ont permis de valider expérimentalement ce travail soient ici remerciés.

Mais développer des outils de CAO n'a de sens que si l'on peut les confronter à la réalité : j'ai eu la chance de partager mon bureau avec deux *designers*, eux même expérimentés en CAO, travaillant sur le démonstrateur du projet européen dans lequel mes recherches s'inscrivent. Je tiens donc à remercier Luis Lucas et Franck Wajsbürt des innombrables et enrichissantes — ou non ! — conversations que nous avons eu, et ce depuis nos lointaines maîtrises... Je suis particulièrement redevable à Franck de son travail sur la transformation des objets symboliques.

En vrac, quelques pensées pour ceux qui ont contribué à la dynamique d'Alliance et à la bonne ambiance de l'équipe lors du démarrage du projet, ainsi qu'à Diana, pour avoir pris calmement mes excentricités durant cette dernière année de rédaction.

Enfin, je suis plus que reconnaissant envers mes parents qui m'ont donné le goût des études et la possibilité matérielle de les mener aussi loin.

Résumé

Cette thèse introduit le concept de bibliothèque portable et détaille la mise en œuvre d'un environnement dédié à leurs développements.

Le premier chapitre présente une méthodologie de dessin de masques symbolique sur grille fixe fine qui permet la portabilité sur un vaste ensemble de procédés de fabrication tout en permettant la construction de blocs complexes par aboutement. Le second chapitre présente des structures de données canoniques pour la représentation des vues physique et structurelle des circuits intégrés *VLSI*. Ces structures de données sont la base sur laquelle a été construite la chaîne de **CAO Alliance**. Les différents concepts attachés aux vues physique et structurelle que l'on désire formaliser sont décrits, en s'attachant à préciser la sémantique de chaque objet de façon non ambiguë. Le troisième chapitre présente un langage dédié au développement de générateurs de blocs paramétrables. Ce langage est une bibliothèque de fonctions du langage **C** qui s'appuient fortement sur les structures de données. Afin de faciliter la mise au point de ces programmes par des concepteurs de circuits, un outil graphique d'aide à la mise au point des logiciels de génération est présenté dans le chapitre 4. Cet outil est constitué d'un éditeur de texte, d'un compilateur **C** ayant pour cible une machine virtuelle, d'un interpréteur, d'un débogueur symbolique, et d'une fenêtre de visualisation de la vue physique. Enfin, la méthode générale de conception d'un générateur paramétrable est illustrée à travers l'exemple d'un générateur de *ROM*.

Mots clef
portabilité technologique
représentations canoniques
génération procédurale
interprétation, mise au point

Table des matières

Liste des figures

Liste des tableaux

Introduction

Le temps écoulé entre la spécification fonctionnelle d'un *ASIC* et la disponibilité des masques permettant la fabrication est critique. L'essentiel des recherches actuelles se focalisent sur les outils de synthèse, qui permettent d'obtenir un schéma en portes à partir d'un comportement spécifié dans un langage de haut niveau. Mais cette approche n'est aisément automatisable que pour des technologies à faible taux d'intégration, FPGA, matrices prédiffusées et bibliothèques de cellules précaractérisées. Dans le futur proche, ces outils de synthèse devront s'interfacer sur des bibliothèques de blocs fonctionnellement plus complexes, afin d'améliorer les performances des circuits, en terme de densité d'intégration ou de temps de cycle[GDWL92]. En effet, l'utilisation de cellules de faible complexité — portes logiques à trois ou quatres entrées — est trop restrictive si l'on désire des performances élevées. Dès que la surface ou les contraintes de temps de cycle deviennent critiques, il faut que les outils de synthèse intègrent des blocs optimisés de complexité plus élevée : banc de registres, additionneurs ou multiplieurs rapides, mémoires. . .

Il est matériellement impossible de fournir des blocs optimisés prédéfinis pour tout l'éventail des paramètres envisageables. L'espace disque, le temps de génération et de vérification, la complexité de gestion, à la fois pour le concepteur de la librairie et l'outil de synthèse imposent le recours aux générateurs de blocs optimisés paramétrables.

Mais concevoir un générateur paramétrable est une tâche ardue, qui coûte cher en temps de développement — de l'ordre de un à deux hommes $\times$ an —.

Un tel investissement n'est rentable que si l'on peut le pérenniser. Pour cela, il faut d'une part que les générateurs soient relativement indépendants du processus de fabrication, et d'autre part que le code du générateur soit portable et non prisonnier d'un système de **CAO** fixé.

La confrontation des concepteurs de librairies à cette double contrainte de portabilité est le sujet de cette thèse.

Avoir une représentation de la vue physique permettant de s'adapter à différents processus de fabrication est une nécessité. Les générateurs par aboutement exigent une prédictabilité absolue de la taille des cellules et de la position des connecteurs. Ils imposent des contraintes spécifiques dans le choix d'une méthode de dessin visant la portabilité sur différentes technologies. Une solution à ce problème essentiel est donnée par l'approche symbolique sur grille fixe sans compactage, présentée dans le chapitre 1.

La méthode de représentation de la vue physique ayant été définie, il faut se doter d'une représentation normalisée de la vue structurelle décrivant la connectique. Ceci aboutit à la définition de structures de données canoniques, pour les deux vues, comme support des outils de génération procédurale. L'effort de formalisation a conduit d'ailleurs à l'utilisation de ces structures dans l'ensemble des outils de la chaîne de **CAO** pour *VLSI* **Alliance**. Les objectifs, la définition et l'implémentation de ces structures sont traités dans le chapitre 2.

La conception de bibliothèques de blocs paramétrables ne peut se faire sans un langage puissant et adapté à la description procédurale. Le choix est de redéfinir un langage et son compilateur, ou d'utiliser un langage existant en l'enrichissant. C'est cette seconde solution qui a été retenue, et qui est détaillée chapitre 3.

Un générateur est un outil logiciel complexe, donc sujet à erreurs de programmation. Afin de faciliter le travail du concepteur lors de l'implémentation de ses algorithmes, un outil d'aide à la mise au point est indispensable. Les particularités de la génération de blocs par programme font que les outils standards ne sont pas adaptés. En effet, s'ils permettent l'inspection des variables et la gestion de l'exécution, ils ne permettent pas une visualisation intuitive, sous forme graphique. C'est pour proposer cette fonctionnalité qu'un outil de mise au point adapté a été développé. Ses divers aspects, de la compilation jusqu'à l'interface graphique, sont exposés dans le chapitre 4.

Un exemple de réalisation de générateur illustre brièvement l'utilisation des outils de description procédurale décrits dans cette thèse. Ce générateur de *ROM* est l'objet du chapitre 5.

Les chapitres 1, 3 et 4 abordant des problèmes distincts, nous avons choisi de présenter l'état de l'art correspondant en début de chaque chapitre.

Ces travaux ont été effectués dans le cadre du projet européen ESPRIT **Integrated Design and Production Sytem**. L'objectif clef du projet est de permettre la conception et la fabrication de circuits de plus d'un million de transistors en environ un mois[IDP90], avec la possibilité de fabriquer indifféremment chez chacun des partenaires fondeurs. Cela passe par la standardisation au niveau européen d'une librarie de cellules commune et d'outils de **CAO**. Les partenaires industriels du projet sont BOSCH, BULL, ES2, PHILIPS, PLESSEY, SGS-THOMSON et SIEMENS.

Chapitre 1

Vue physique symbolique

1.1 Introduction

L'investissement en temps nécessaire au dessin des masques des composants d'une bibliothèque est considérable. Si c'est déjà vrai pour une bibliothèque de cellules précaractérisées — quelques dizaines de cellules —, c'est encore plus vrai pour une bibliothèque de générateurs — plus de cinq cents cellules pour les générateurs de la chaîne de **CAO Alliance** —. La conception de bibliothèques portables est donc essentielle pour éviter le fastidieux travail de dessin de cellules adaptées à chaque technologie. En effet, l'édition de cellules reste une des dernières activités de la conception *VLSI* qui soit encore complètement manuelle, car la nécessité conjuguée de l'intégration maximale et de la performance électrique sont loin d'être atteintes par les compilateurs de cellules. De plus, ces compilateurs ne savent travailler que dans un contexte très bien défini, comme le *gate matrix*[LL80], incompatible avec les contraintes des circuits optimisés assemblés par aboutement dans les deux directions x et y.

Les contraintes de la portabilité sont assez différentes selon les technologies d'assemblage :

- les bibliothèques de cellules précaractérisées, utilisables avec des outils de placement et routage pour *standard cells*. Dans les technologies à deux niveaux de métal, elles sont assemblées en bandes séparées par des canaux de routage. Dans ce cas, les cellules doivent toutes avoir la même hauteur, partager les lignes d'alimentations et avoir la même politique de placement des entrées/sorties.

- les bibliothèques de cellules dédiées à un générateur. Les cellules sont utilisées par un programme d'assemblage qui permet de créer des blocs paramétrables. Ce sont en quelque sorte les briques de base de bibliothèques dédiées.
  Dans ce cas, les cellules ont des tailles variables. Elles doivent pouvoir s'abouter dans les deux directions x et y dans un assemblage de type mosaïque . Ces contraintes sont dites de *pitch matching* dans la littérature.

- les bibliothèques de macro-blocs figés, comme des cœurs de microprocesseurs. Ces *mega-cells* comportent un mélange des deux styles précédents, plus du routage entre blocs.

Si l'on vise chacun de ces types, comme c'est notre cas, il est clair qu'il faut que les changements de technologie se fassent de manière à conserver tous les aboutements, que ce soit au niveau des cellules ou du routage.

1.2 Méthodes de dessin visant la portabilité : état de l'art

Diverses méthodes sont utilisées pour la description topologique des masques. La méthode consistant à dessiner en utilisant des règles au micron d'un unique procédé de fabrication n'entre pas dans la discussion. Notons néanmoins que c'est ainsi que la densité d'intégration maximum sera théoriquement atteinte.

Trois grandes familles d'approches pour la conception symbolique sont usuellement distinguées[Gaj88] :

- le dessin sur grille fixe. Le pas de grille est indépendant des objets placés et de leur voisinage. La grille physique est homothétique à la grille symbolique.
- le dessin sur grille virtuelle. Le pas de grille physique est fonction de la plus forte contrainte entre les éléments de deux lignes ou colonnes adjacentes. Le pas de grille physique est donc irrégulier, mais la grille est toujours constituée de droites.
- le dessin sur grille relative. Les objets sont compactés et désalignés. La grille physique est constituée de lignes brisées.

La figure 1.1 montre l'aspect des grilles physiques résultantes pour ces trois familles (d'après [Nav92]). Notons toutefois que la valeur du pas de grille, qui pour la clarté du dessin est du même ordre de grandeur est très variable suivant les approches choisies. Pour les méthodes liées à la compaction, la configuration initiale fournie par le concepteur

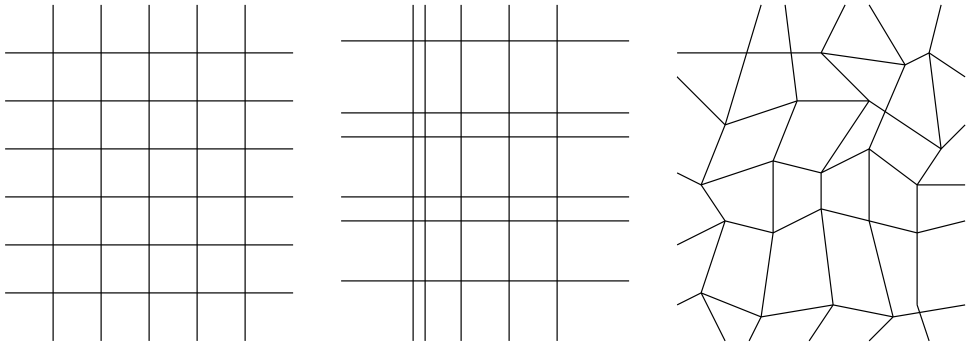


Figure 1.1: Grilles physiques résultant d'une approche sur grille fixe, virtuelle et relative.

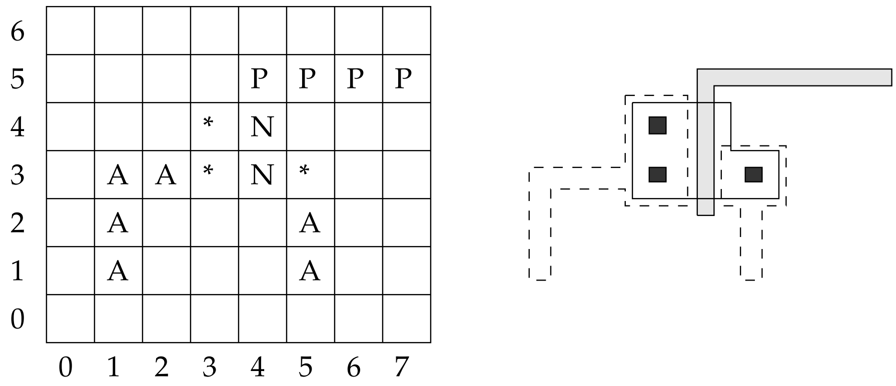
est fondamentale — tout autant que pour l'approche à grille fixe —. En effet, le domaine d'influence du compacteur est très local, et un dessin symbolique mauvais donnera une cellule au micron mauvaise.

Quelques méthodes, ainsi que leurs conséquences du point de vue des bibliothèques portables, sont étudiées.

1.2.1 Dessin sur grille fixe grossière

L'introduction de la notion de symbole pour la représentation des masques a été utilisée par la société ROCKWELL[Lar71], en 1971. Cette approche utilise une grille grossière, *coarse grid*, dont le pas est défini par la plus défavorable des configurations dessinables. C'est en général la distance centre à centre entre deux contacts qui est choisie comme telle, ce qui donne un pas de grille symbolique de l'ordre de 30 fois la précision de la machine à graver. (Par exemple, une technologie 1.2 μm a une résolution de .15 μm, pour une distance centre à centre entre contacts de 4.2 μm.) Chaque carré de grille est éventuellement occupé par un symbole, qui représente soit un niveau de masque, soit un contact.

Les outils **MASK**[Lar71] et **SLIC**[GN76] sont représentatifs de cette méthode, où la saisie des masques se fait par l'utilisation de lettres auquel un sens est attaché. Chaque carré formé par la grille est éventuellement occupé par un caractère qui, lors de l'expansion, va se transformer en masque ou ensemble de masques, et prendre une taille définie par la technologie cible. L'illustration de ce principe est donnée figure 1.2 par un exemple classique. La simplicité du dessin se paye par une perte de surface importante, due à la



(a) (b)

Figure 1.2: Un exemple de dessin sur grille grossière, (d'après [WE85].)
(a) saisie symbolique textuelle,
(b) rectangles après expansion.

dispersion des contraintes de distance pour les différents niveaux dans un même processus **CMOS**. La précarité de la saisie ainsi que la mauvaise densité obtenue ont fait tomber cette approche en désuétude.

1.2.2 Dessin au λ

Introduit dans les années 1980 par Mead et Conway[MC80], l'approche se base sur un paramètre d'échelle unique, le λ. Le dessin se fait grâce à des polygones rectilinéaires, avec des règles de dessin simplifiées. Tous les points constituants les polygones sont positionnés sur une grille de pas 1 λ dans les deux directions. La valeur du λ est de l'ordre de la longueur de grille minimum du transistor de la technologie considérée. La grille est dite *fine*, et vaut entre cinq et dix fois la résolution minimale de la ligne de fabrication.

Les règles de dessin au λ sont obtenues à partir d'un ensemble de règles réelles au μm, par changement d'échelle et arrondi aux valeurs entières supérieures. La valeur du λ est déterminée pour une technologie donnée, et chacune des coordonnées du dessin est multipliée par cette valeur pour donner la coordonnée réelle. Ceci permet d'englober plusieurs technologies en calculant la valeur minimale du λ qui ne cause pas de violation pour l'ensemble des règles réelles considérées. (Il est toujours possible de satisfaire une technologie donnée en choisissant un λ assez grand.)
L'avantage de la méthode tient en sa simplicité :

- les règles de dessins sont exprimées par un petit nombre d'entiers, donc simples à mémoriser.
- la formule de passage de la grille symbolique à la grille réelle est une simple homothétie. Le λ étant un nombre entier de pas de grille réelle, il n'y a pas de problème d'arrondi lors de l'homothétie.

Le rapport des distances est constant quelle que soit la technologie cible, ce qui garantit la conservation des alignements dans toutes les directions. Mais cette transformation simpliste ne peut être faite sans perte inacceptable de surface que pour des technologies assez semblables. L'inexactitude essentielle tient à l'utilisation du λ comme paramètre unique. Ceci suppose que les distances bord à bord varient peu entre les technologies, et que l'on peut faire une transformation homotétique des masques. Ceci s'avère malheureusement faux dans la réalité.

Afin de contourner ce problème, tout en conservant la simplicité de l'approche du point de vue du concepteur, des raffinements ont été ajoutés dans l'opération de transformation au micron. On commence par faire une mise à l'échelle, *linear scaling*, suivie d'opérations d'expansion, *bloat*, ou de réduction, *shrink*, spécifiques à chacun des niveaux de masques. C'est cette technique qui est utilisée dans le compilateur de silicium **GENESIL**[CM88].

1.2.3 *Stick* suivi de compaction

Le premier algorithme de compaction monodimensionelle a été présenté en 1970 par Ackers[AGR70]. Les idées de base de la *compression ridge method* sont encore exploitées aujourd'hui[Sch87]. Depuis, un grand nombre d'investigations ont eu lieu, aussi bien sur la manière de saisir les masques que sur les algorithmes de compaction. La saisie de masques sous un éditeur graphique *stick* se fait par le dessin de fils sans épaisseur, la création automatique de contacts lorsqu'un fil se termine sur un autre fil, et de transistors quand du polysilicium croise de la diffusion, par exemple. Il n'y a pas forcément de grille contraignant la position des points. Des attributs peuvent être attachés aux éléments, telle la largeur des fils. Une phase de compaction automatique qui transforme les éléments et tente de les placer au minimum des règles réelles est appliquée. Les approches *virtuelle*, comme celle implantée dans **MULGA**[AW83], et *relative*, celle que l'on retrouve dans **SLIM**[Dun80], relèvent toutes deux de la compaction, c'est l'algorithme appliqué qui les différencie. Une étude comparative des différentes approches de la compaction a été faite par Boyer[Boy88].

Cette technique est attractive pour une cellule de base car elle garantit la validité des règles par construction.

Mais elle est simplement inutilisable dans un contexte où les contraintes d'aboutements de cellules imposent la position des connecteurs dans les deux directions, comme dans les blocs optimisés.

- compaction hiérarchique. La position des connecteurs des cellules après compaction n'est pas prévisible, et peut être très variable suivant les technologies. Une tentative de résolution de ce problème est proposée par Liao[LW83], en permettant au concepteur d'imposer des contraintes sur les tailles et les positions dans les éléments du dessin. Mais les propagations de contraintes dues aux aboutements d'une cellule avec de multiples autres cellules peuvent créer des cycles dans le graphe des contraintes, et le compactage peut ne jamais converger[BV93]. De plus, cette approche exige que les cellules soient disjointes, c'est-à-dire qu'elles respectent des demi-règles de dessin par rapport à leurs enveloppes[Ute91]. La compaction hiérarchique reste un problème ouvert, et est l'objet de nombreuses recherches[BV92][Mar90].
- compaction à plat . Le problème de la compaction dans les directions x et y simultanément est *NP* complet. La compaction dans une direction, puis l'autre, est quadratique. Ceci rend la méthode inutilisable pour traiter en une fois des blocs de taille importante. L'utilisation conjointe d'heuristiques et du partitionnement tente de résoudre ce problème[DMH$^+$93].

Le résultat d'une génération d'un bloc constitué de cellules issues du compactage n'est pas technologiquement indépendant, ce qui en interdit un usage hiérarchique dans les *megacells*.

L'utilisation de ces méthodes de compaction est bien adaptée aux cellules de bibliothèques de cellules précaractérisées[YCD$^+$93].

1.2.4 Dessin sur grille fixe grossière localement déformable

L'utilisation d'une grille fixe couplée à des techniques de compaction locale est l'idée de base de l'outil **PRÉFORME**[DNJ90][Nav92]. La saisie des objets se fait sous un éditeur *stick* contraignant leur position sur une grille dont le pas est souvent choisi comme la distance centre à centre entre deux contacts. Les fils sont horizontaux ou verticaux, et les transistors sont crées par croisement de diffusion et de polysilicium.

Une phase d'expansion permet de déformer localement les fils, et de déplacer dans une zone d'influence prédéfinie les objets du dessin. Le nombre de sites utilisables pour le positionnement des objets étant réduit, c'est l'outil qui va se permettre de les désaligner pour exploiter une grille plus fine, inaccessible à l'utilisateur lors de la saisie.

Cette approche mixte cherche à concilier les avantages de la grille fixe et de la compaction. Ceci doit garantir une conservation des alignements, et permettre une construction hiérarchique en rendant des zones non déformables autour des aboutements, un peu à la manière des contraintes des compacteurs. Cette approche vise à atteindre une densité plus importante et à simplifier la saisie des cellules en limitant par construction le nombre de sites occupables.

Cette approche pose les problèmes suivants :

- à cause de la déformabilité, et de la propagation de contraintes en résultant, les règles de dessin sont dépendantes du contexte. Les règles symboliques ne sont pas uniquement des règles de distances à deux corps. Il faut intégrer des contraintes de plus haut niveau comme : on ne peut aligner plus de 3 contacts sur 3 pas de grille consécutifs. En conséquence, le concepteur doit mémoriser des règles en plus des règles de distances.

- la mise en œuvre d'un tel outil dans un contexte hiérarchique avec aboutement en x et y exige que l'on considère toujours le pire cas sur les contraintes d'aboutabilité entre cellules. La taille de la cellule étant déterminée par son interface, la compaction interne a pour seul but de garantir le respect de cette taille. La perte d'un pas de grille dans ces conditions n'est pas négligeable, car cela correspond à la distance centre à centre entre deux contacts.

- tout système symbolique est dépendant des paramètres de la technologie concernant le comportement des transistors, car le rapport I_P/I_N varie suivant les procédés de fabrication. Pour garantir le respect des règles de dessin, le système peut être amené à modifier les caractéristiques géométriques des transistors saisis si la place disponible est insuffisante. Néanmoins, en réservant par défaut un nombre suffisant de sites, il est possible d'adapter assez précisement le dimensionnement des transistors à une technologie lors de l'expansion.

1.3 Dessin sur grille fine sans compaction

Pour pallier les difficultés des méthodes présentées, nous avons opté pour une méthode de dessin sur grille fine sans compaction. L'absence de compaction est nécessaire pour simplifier l'écriture des générateurs par aboutement.

1.3.1 Principes de la méthode

Cette méthode se fonde sur une approche statistique permettant de définir les paramètres pertinents qui varient peu avec les technologies[GL90]. Ces études, menées sur un échantillon d'une vingtaine de technologies à deux niveaux de métal et un niveau de polysilicium avec des longueurs de grilles de 2 μm à 0.5 μm ont conclu que les paramètres dont la variation est la plus homogène, c'est-à-dire quasi-homothétique, sont les distances centre à centre , le *pitch*, pour un niveau de masque donné. Ce résultat explique les raisons de la faiblesse de l'approche au λ classique, car elle fait implicitement l'hypothèse que c'est la distance bord à bord qui est invariante à un facteur d'échelle près. À partir de cet échantillon de technologies et de la constatation que le dessin se fait essentiellement avec des fils de largeur minimale, on peut définir un jeu de règles de dessin symboliques au λ. La valeur du λ est de l'ordre de la longueur de grille d'un transistor minimum pour un processus de fabrication donné.

La méthode symbolique sur grille fixe sans compaction peut être vue comme une généralisation du dessin au λ, car c'est encore la pire des règles réelle de l'échantillon qui va imposer la valeur du λ. Mais la dispersion est beaucoup plus faible avec les *pitches* qu'avec les distances bord à bord. Elle hérite des avantages de cette approche tout en palliant certains de ses inconvénients. Elle s'en distingue par les points suivants :

- le dessin repose sur l'utilisation d'objets structurés, des *symboles*, auxquels une sémantique forte est associée — on ne peut pas créer de transistors en faisant se croiser de la diffusion et du polysilicium par exemple —. Les symboles, définis dans le paragraphe 2.3.2, sont soit des mono-points, contacts, références et connecteurs, soit des bi-points orientés, segments et transistors, et sont placés sur les nœuds d'une grille isotrope. Le dessin se fait avec un jeu de masques simplifié et des règles symboliques exprimées par des petits entiers. L'utilisation de ces primitives évite les dessins douteux, et supprime ainsi d'éventuels problèmes électriques.

- dans la phase de traduction du dessin symbolique vers le procédé de fabrication cible, on dispose d'un degré de liberté supplémentaire par rapport au λ classique. En effet, la distance entre les objets et la taille des objets peuvent être ajustés indépendamment. Par exemple, la distance entre deux fils dépend de leur distance centre à centre sur la grille, c'est à dire de la valeur du λ, alors que la largeur minimale de chaque fil est définie par un paramètre technologique indépendant pour chaque type de fil.

- chaque symbole est macro-généré suivant les desiderata du technologue. Typiquement, la transformation d'un morceau de diffusion sera différente pour un fil, un drain de transistor, ou un contact de polarisation du substrat. Il est également possible de jouer sur la largeur des grilles de transistors en fonction du rapport des mobilités électroniques des diffusions pour un procédé donné.

Cette méthode de dessin symbolique sur grille fine est utilisée industriellement par la compagnie BULL qui ne possède pas de ligne de fabrication. Elle a été adoptée pour les bibliothèques portables de la chaîne **Alliance** développée au laboratoire **MASI**.

1.3.2 Restrictions sur la portabilité

Le dessin symbolique sur grille fixe impose des contraintes restrictives sur la conception. Ces contraintes visent à garantir un éventail plus grand de technologies cibles, un nivellement par le bas en quelque sorte :

- les règles de dessin s'entendent avec des distances de *manhattan*. Les segments ne peuvent être qu'horizontaux ou verticaux. Les segments inclinés, y compris à 45^{o}, sont interdits.

- les technologies visées possèdent un niveau de polysilicium et deux niveaux de métal.

- les transistors coudés doivent avoir la longueur de grille symbolique minimale — l'utilisation de coudes dans des transistors affaiblis n'ayant pas d'utilité réelle, cette contrainte est purement formelle —.

- les contacts de polarisation du substrat ne peuvent pas être placés sur les sources de transistors.

- il faut respecter une distance minimum entre les niveaux via et polysilicium, car certaines technologies l'imposent.

- l'empilement des contacts est interdit, car seules un nombre restreint de technologies permettent les *stacked via* aujourd'hui.

Si le premier point est intrinsèque à la méthode, les autres sont des règles visant à garantir la portabilité effective. Si toutes les technologies envisagées possédaient des *stacked via*, alors la dernière règle pourrait être omise lors de la conception de cellules.

Tout changement drastique de la technologie exige que les cellules de bibliothèques soient redessinées, et ce quelle que soit la méthode adoptée. L'introduction d'un troisième niveau de métal et des empilements de contacts remet en cause les habitudes de conception et les règles de dessin à utiliser, mais pas la méthode en elle même s'il est vrai que les distances centre à centre entre les objets continuent à varier relativement linéairement sur un ensemble de technologies.

La perte de surface attendue, après transformation vers la technologie cible, par rapport à un dessin aux règles réelles est de 10% à 20% d'après [CM88][GL90]. Néanmoins, puisque le dessin des cellules est fait une unique fois, le concepteur peut passer plus de temps à les optimiser, en exploitant de manière judicieuse la finesse de la grille de dessin.

1.3.3 Représentation des objets symboliques en rectangles

Conceptuellement puissante pour faciliter la conception comme pour assurer la portabilité, la notion de symbole doit laisser place aux rectangles pour les raisons suivantes :

1. permettre une visualisation intuitive pour le concepteur.
2. permettre la vérification des règles de dessin symboliques avec des outils de vérification commerciaux classiques. Les outils existant fonctionnent sur des rectangles ou des polygones, car les fondeurs expriment leurs règles par rapport à des bords de rectangles.
3. permettre l'utilisation d'extracteurs classiques fournissant une *netlist* de transistors et de capacités.
4. fournir des fichiers **cif** ou **gdsII** adaptés à la technologie cible.

Pour les points 1, 2 et 3, une technologie cible virtuelle a été définie. L'expansion lors de la saisie est illustrée figure 1.3.

Les détails de la transformation du symbolique vers une technologie cible et la paramétrisation du fichier technologique sont explicités dans l'annexe A. Notons que cette transformation est hiérarchique et permet donc de traiter les gros circuits.

Les étapes principales de cette transformation sont brièvement décrites ci-dessous. Le résultat de l'expansion ne sera valide, du point de vue des règles au micron, que si le circuit est validé pour les règles symboliques.

- la valeur du pas de grille symbolique — aussi appelé λ pour simplifier l'écriture — est calculée en étudiant les règles au micron du processus de fabrication. Le λ est toujours un nombre entier de pas de grille physique.
- la grille de définition des objets est réduite de façon homothétique. La position relative du centre des symboles est donc conservée.

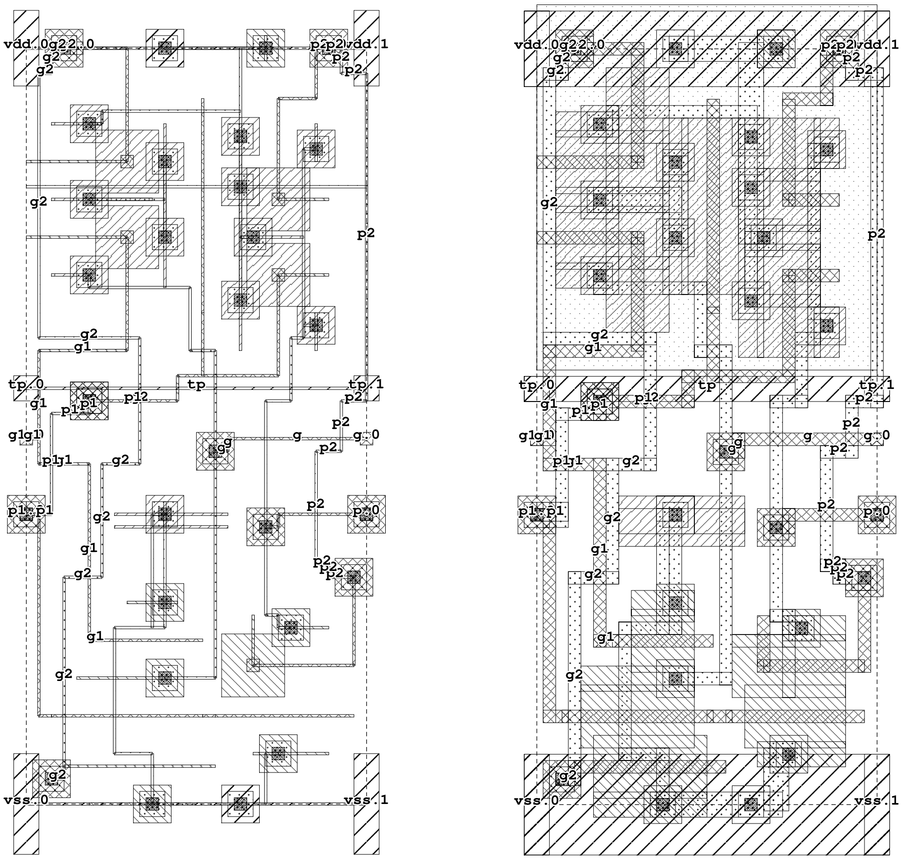


Figure 1.3: Expansion des objets lors de la saisie symbolique.

- chacun des symboles est expansé, en prenant en compte son type et des informations technologiques fournies dans un fichier de paramètres. La transformation la plus complexe est celle des segments, dont la largeur est calculée comme suit :

  $$W_{réel} = (W_{symbolique} - W_{minimum\ symbolique})\ \lambda + W_{minimum\ réel}$$

  On constate sur cette formule qu'une largeur minimum symbolique doit être définie pour les niveaux éditables, et que seuls les fils de largeur non minimale dépendent de la valeur du λ.

La complexité algorithmique est jusqu'ici linéaire. Néanmoins, et afin d'obtenir une bonne densité, on autorise l'expansion à créer deux types de violations de règles de dessin que l'on va corriger systématiquement ensuite.

- *notches* dus aux contacts.
- distances minimum entre les zones d'implantation.

La correction de ces erreurs impose une phase de fusion des rectangles, d'un complexité quadratique que l'on réduit en partionnant en moyenne à $O(n^{\frac{3}{2}})$.

Notons qu'il n'est pas possible d'ajuster indépendamment et automatiquement le rapport de taille des transistors P et N , ce qui peut conduire à des dégradations de perfomances sur des technologies ou le rapport I_P/I_N est très différent de celui supposé lors de la saisie.

La figure 1.4 illustre les mécanismes mis en œuvre lors de la macro-génération des objets terminaux de la conception. Les croix et lignes représentent les informations topolo-

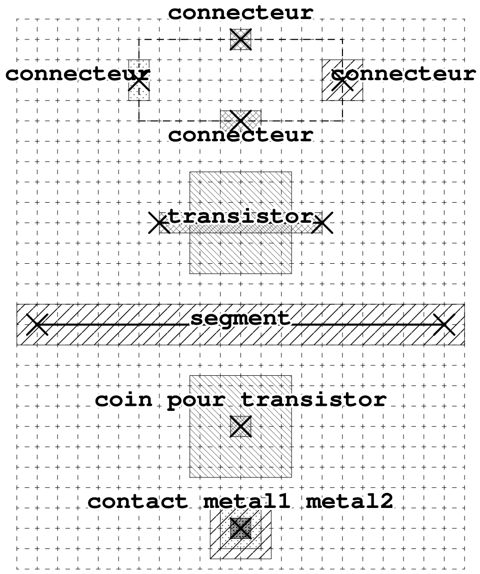


Figure 1.4: Expansion des objets symboliques en rectangles.

giques symboliques, les aires hachurées en étant le résultat après expansion, tel qu'il sera visible sous un éditeur graphique symbolique.

Les figures 1.5 et 1.6 montrent la transformation de deux transistors et d'un *via* vers une technologie 1.2 μm. On peut constater une importante différence entre les

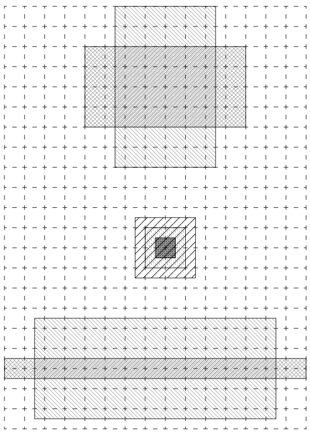


Figure 1.5: Objets transformés en rectangles symboliques .

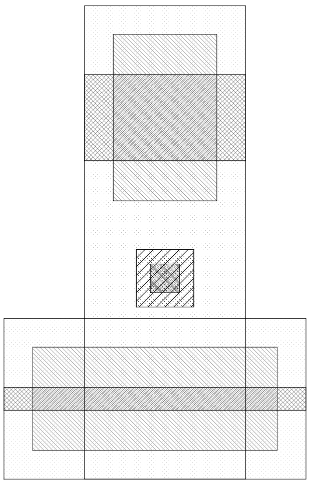


Figure 1.6: Les mêmes objets en une technologie 1.2 μm.

représentations symboliques et réelle.

- le contact métal1-métal2 est remarquable. On constate que la tranformation des niveaux dans un *via* n'est pas du tout homothétique pour ce procédé de fabrication.

- les diffusions des transistors *N* ont été transformées en zones actives et zones d'implantation *N*. Les zones d'implantation étant trop proches pour la technologie visée, elles ont été fusionnées par l'outil de traduction.

(a)
(b)
(c)

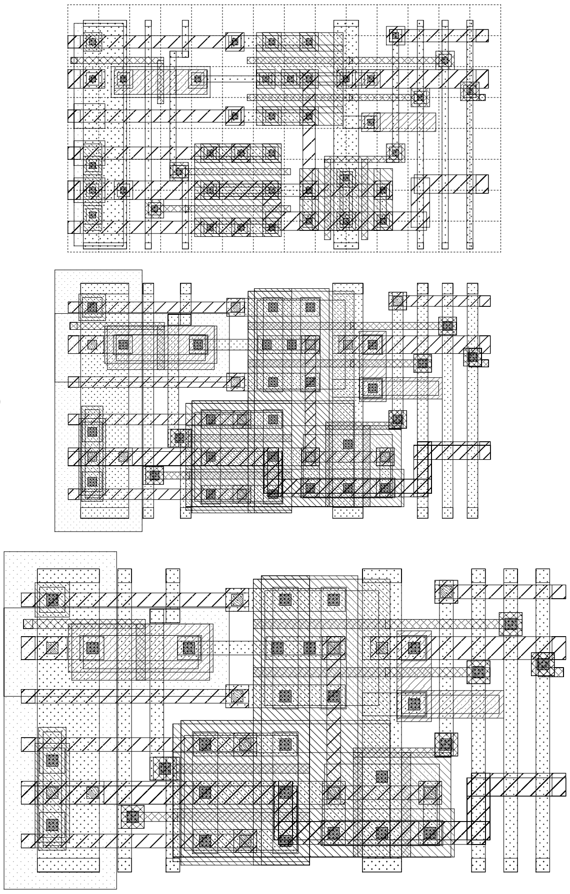


Figure 1.7: Transformation du symbolique vers le réel.
(a) Figure vue sous l'éditeur graphique symbolique.
(b) Résultat de la migration vers une technologie 1.2 μm, le λ valant 1.05 μm.
(c) Résultat de la migration vers une technologie 1.5 μm, le λ valant 1.4 μm.

La figure 1.7 montre la transformation d'une cellule symbolique optimisée expansée vers deux technologies réelles. Dans son contexte d'utilisation, cette cellule a des contraintes d'aboutement verticales et horizontales. Les règles de dessin symboliques sont détaillées dans [Ze91].

Une des qualités intéressante de notre approche est la possibilité de substituer lors de la traduction des masques une cellule ou un macro-bloc par un objet équivalent dessiné sur la technologie cible. Pour cela, il suffit de dessiner un fantôme symbolique qui a le même encombrement et des connecteurs recouvrant ceux du bloc réel une fois transformé pour la technologie cible. L'annexe A.6 illustre le procédé pour le remplacement d'une cellule de base.

1.4 Comparaisons avec d'autres méthodes symboliques

L'utilisation de la compaction ne permet pas la construction par aboutement dès que l'on vise de multiples technologies, car les méthodes de compaction hiérarchique sont très coûteuses à mettre en œuvre et échouent parfois lorsque l'on doit garantir des coïncidences, comme on l'a vu au paragraphe 1.2.3.

De plus, la méthode symbolique sur grille fixe a des qualités importantes pour la génération procédurale. En effet, il est nécessaire de pouvoir dessiner par programme des fils entre cellules, entre les connecteurs mais aussi entre des points internes aux cellules, ou de venir y placer des contacts, car ceci permet de limiter le nombre de cellules de base pour des décodeurs par exemple. Cette technique n'est pas utilisable si l'opération de transformation de la représentation symbolique en masques produit des désalignements.

L'autre problème lié aux méthodes de dessin symboliques utilisant la compaction est que tous les outils qui manipulent la vue *layout* doivent être paramétrables pour une technologie donnée, car ils se fondent sur des cellules déjà dépendantes du processus de fabrication. Dans notre approche, toute la conception du circuit manipule des objets symbolique, jusqu'au routage final. Ceci permet de profiter de la technologie la plus récente lors de l'achèvement du projet, ou de choisir un fondeur en fonction des moyens disponibles. L'outil réalisant l'expansion vers la technologie cible étant hiérarchique, il est possible de traiter de très gros circuits.

1.5 Résultats pratiques

L'utilisation du dessin symbolique sur grille fixe sans compaction a permis d'obtenir des résultats très encourageants :

- environ six cents cellules ont été dessinées en quatre ans au laboratoire en uti-lisant cette méthode. Ceci prouve expérimentalement que le niveau d'abstraction de l'approche permet un développement rapide des bibliothèques.
- des circuits d'enseignement et de recherche développés au laboratoire MASI ont été fabriqués sur les technologies ES2 qui nous sont accessibles, nommément 2 μm, 1.5 μm et 1.2 μm, et ce avec succès. Le tableau 1.1 ci-dessous présente quelques chiffres concernant des blocs construits par aboutement, pour une technologie 1 μm, paramètres pire cas à 70°. (La colonne délai est le temps d'accès en lecture pour les éléments séquentiels.)

Tableau 1.1: Données qualitatives sur quelques générateurs.

bloc optimisé	transistors	densité (Kt/mm^2)	délai (ns)
additionneur 32 *bits*	1513	4.7	10.5
multiplieur 32 × 32 *bits*	23875	5.4	52.0
décaleur à barillet 32 *bits*	2828	2.5	9.5
banc de registres de 32 mots de 32 *bits*	11656	5.3	8.0
ROM de 64 Kilo *bits*	75048	14.0	8.5
RAM de 1Kilo mots de 32 *bits*	201686	11.0	11.5
FIFO 32 mots de 32 *bits*	12176	6.0	8.2

- cette méthode a été utilisée dans d'autres contextes industriels, et l'outil de transformation a alors été paramétré par l'utilisateur, celui-ci ne pouvant dévoiler ses informations technologiques :
  - la partie opérative d'un processeur de traitement du signal destiné à l'évaluation des outils de bas niveau du laboratoire a été développée par SGS-THOMSON, et traduite en technologie 0.7 μm.
  - de nombreux circuits produits par BULL, notamment dans le cadre du projet **Auriga**, ont été réalisés chez différents fondeurs, pour des technologies entre 1.2 μm et 0.6 μm.
  - le processeur **StaCS**, démonstrateur de la seconde phase d'**IDPS** d'une complexité de 875000 transistors, a été entièrement implémenté avec cette méthode de dessin. Deux jeux de masques ont été produit pour deux technologies cibles : PHILIPS 0.8 μm et SGS-THOMSON 0.8 μm.

1.6 Conclusion

Ce chapitre a présenté une méthode de dessin symbolique sans compaction qui assure une réelle indépendance technologique et une bonne productivité pour les blocs construits par aboutement.

Les contraintes de ces blocs sont telles que l'utilisation de compacteurs est impossible. Garantir des juxtapositions dans les deux directions x et y sur un ensemble de cellules s'aboutant de multiples façons implique la définition des contraintes entre chacun des connecteurs de chacune des cellules avoisinantes. Cette approche est très coûteuse en temps de développement, et il est parfois impossible de faire converger le compacteur.

D'autres outils permettent de figer l'enveloppe, rendant immuable l'encombrement et la position des connecteurs, mais alors on ne voit plus très clairement le gain en surface, car si l'enveloppe n'est pas modifiable, compacter l'intérieur ne rend pas les cellules plus denses pour autant.

Des diverses approches utilisant le λ, celle utilisant des techniques d'érosion et de dilatation est satisfaisante du point de vue portabilité, mais pas assez abstraite pour réduire de façon significative le temps de dessin des cellules de base.

L'expérimentation de notre méthode symbolique sans compaction depuis quatre ans donne des résultats satisfaisants dans les domaines suivants :

- temps de développement de bibliothèques de cellules : dessiner six cents cellules de base en quatre ans est certainement une performance dans un cadre universitaire.
- développement de générateurs de blocs optimisés : sept générateurs de blocs ont été construits en quatre ans.
- portabilité effective sur des technologies variées : les résultats du paragraphe 1.5 le prouvent. De plus, le processus de transformation est relativement simple puisqu'entièrement automatique.
- densités obtenues : l'utilisation des aboutements dans les blocs optimisés mais aussi dans les chemins de données utilisant des cellules précaractérisées et des blocs optimisés permettent d'obtenir des densités élevés[Cha92], (jusqu'à 5000 Transistors par mm^2 dans une technologie 1 μm pour les chemins de données).
- performances électriques des circuits réalisés : les simulations sur le chemin critique extrait des générateurs développés au laboratoire fournissent des délais équivalents à ceux obtenus avec des générateurs optimisés pour un *process* particulier.

La méthode a cependant des limites : aujourd'hui, le développement de bibliothèques portables va vraisemblablement de nouveau être nécessaire pour profiter des avancées des lignes de fabrication et des technologies émergentes :

- les technologies **CMOS** 0.5 μm utilisent trois niveaux de métal et généralisent les empilements de *vias*. Pour tirer parti de ces changements radicaux, il va falloir faire évoluer les règles de dessin symbolique, redessiner des bibliothèques. Notons que l'augmentation du nombre de niveaux utilisables pour les interconnexions oblige également à repenser totalement les outils de placement/routage.
- la technologie **AsGa** MESFET est une cible de choix pour le développement de bibliothèques optimisées. L'adaptation des objets symboliques à la technologie **AsGa** MESFET a nécessité uniquement la redéfinition des objets transistors. Ceci montre la généralité de l'approche proposée.

Le prix à payer est la définition d'un jeu de règles de dessin générique pour chaque famille de technologies, ce qui ne remet pas en cause la généralité de l'approche.

Chapitre 2

Structures de données canoniques pour la représentation des vues physique et structurelle des circuits intégrés *VLSI*

2.1 Introduction

Un des aspects les moins satisfaisants pour le développement de circuits est la diversité des formats de représentation de l'information, formats souvent liés à un environnement de conception spécifique. L'objectif de ce chapitre est de présenter une plate-forme de développement pour outils de **CAO** qui tente de résoudre ce problème en prenant explicitement en compte le besoin d'uniformisation. Cette plate-forme repose sur une structure de données orientée objet destinée à fournir un modèle non ambigu des vues physique et structurelle des circuits intégrés. Le besoin d'interopérabilité avec les systèmes de **CAO** externes exige une définition canonique des objets de la conception.

Ainsi, les structures de données présentées dans ce chapitre ont un double objectif :

- rendre les différents outils de **CAO** — générateurs, routeurs, extracteurs... — indépendants des formats de représentation externes : il y a un découplage complet entre les outils qui ne manipulent que des objets en mémoire et les fonctions de lecture et d'écriture des fichiers sur disques. Seules ces fonctions sont dépendantes du format de représentation externe choisi.
- fournir aux outils de **CAO** des objets non ambigus à sémantiques fixes, garantissant un certain niveau de cohérence de façon à faciliter la communication entre outils.

Initialement définies comme support pour les outils de génération procédurale — d'où leur nom MBK, *make block* —, ces structures de données à sémantique fixe ont évolué en plate-forme de développement pour l'ensemble des outils de la chaîne de **CAO Alliance**.

Ce chapitre présente tout d'abord les choix de principes sur lesquels repose la structure de données. Ensuite sont exposés objet par objet, attribut par attribut, chacun des éléments qui la constituent. Quoiqu'un peu fastidieuse, cette énumération permet de clarifier la sémantique des objets, ce qui est indispensable aussi bien pour le développeur d'outils de **CAO** que pour le concepteur de circuits.

2.2 Principes

Les principes ayant conduit à la définition de ces structures de données sont les suivants :

1. **fixer la sémantique.** Cette stratégie consiste à donner à chaque attribut d'un objet de la structure de données un nom et une signification unique et précise, invariable au cours du temps et indépendante des outils qui manipulent ces objets. Ce choix de principe garantit que l'échange d'information entre différents outils ne repose pas sur des conventions tacites, car la structure de données est complètement explicite. Le nombre et la diversité des outils constituants une chaîne de **CAO** sont tels qu'il n'est pas envisageable ni souhaitable de définir une interface pour chaque paire d'outil.

   À la différence des structures de données usuelles, celles présentées ici ne sont pas dédiées à l'implémentation d'un algorithme particulier mais à une représentation canonique des objets. Les concepteurs de circuits se heurtent quotidiennement à la diversité des outils de **CAO** et des formats de représentation de l'information. Le nombre de formats standards pour la représentation des *netlist*s en est l'illustration : **edif**, **vhdl**, **spice**, *etc*. Définir un modèle de données canonique permet de clarifier le sens que l'on donne aux informations présentes dans ces fichiers. Cette formalisation se base sur l'expérience des concepteurs de circuits et sur l'étude des chaînes de **CAO** commerciales.

2. **séparer la vue physique de la vue logique.** Parmi les outils de **CAO**, certains utilisent uniquement la vue logique — simulateur, comparateur de *netlists* —, d'autres uniquement la vue physique — éditeur graphique —, d'autres encore les deux — routeurs, extracteurs —. La plupart des formats de fichiers standards ne savent représenter qu'une vue. Ces constatations nous incitent à représenter les vues physique et structurelle sur deux structures disjointes dont le couplage est limité aux seuls noms d'entités.

3. **représenter la hiérarchie.** La complexité des circuits actuels nécessite l'usage de la hiérarchie. Cette notion permet de décrire un bloc père en fonction de blocs fils instanciés. Les outils basés sur cette structure visent des circuits contenant plus d'un million de transistors, circuits que l'on ne peut pas décrire sans ce concept puissant. Ceci passe par une définition claire de l'interface d'un bloc pour les différentes vues représentées. Cette interface contient l'information minimale attachée à un bloc lorsqu'il est instancié.

4. **encapsuler l'accès à la structure.** Afin de garantir un certain niveau de cohérence et un respect de la sémantique des éléments de la structure, l'accès se fait à travers un ensemble de fonctions. Les règles de cohérence sont explicitées lors de l'exposé des

éléments des structures, aux paragraphes 2.3.2 et 2.4.2. La connaissance de ces règles permet aux outils de récupérer la structure dans un état parfaitement défini.

Fournir des fonctions d'encapsulation permet également de simplifier la programmation, puisqu'un ajout ou une recherche se font par un simple appel de fonction. Ce choix est semblable à celui adopté pour la librairie standard du **C** où des fonctions permettent la manipulation des chaînes de caractères.

5. **mettre à la disposition des concepteurs d'outils de CAO des fonctions de haut niveau.** Certaines fonctionnalités complexes — mise à plat de la hiérarchie, construction du dual d'un graphe d'interconnexions — sont utilisées par beaucoup d'outils pour préparer les données avant de les traiter. L'idée est de proposer ce type de fonctions sur les structures canoniques afin d'éviter de développer plusieurs fois la même fonctionnalité. Ainsi, l'attention des développeurs peut se focaliser sur les algorithmes spécifiques à leur application.

6. **masquer la distinction mémoire/disque.** Le développeur n'a pas à se préoccuper du chargement à partir du disque, et il n'a même pas besoin de connaître l'existence de fichiers. C'est la fonction d'accès aux modèles qui gère implicitement le chargement en mémoire des fichiers afin d'assurer d'une part la cohérence de la structure, et d'autre part la simplicité du modèle tout en mémoire . Le mécanisme est le suivant : lorsque l'accès à un modèle est requis, il est recherché par son nom en mémoire. Si le modèle n'est pas présent, alors une fonction de lecture de fichier, prenant comme argument le nom du modèle, est activée. Cette fonction est paramétrée extérieurement par l'utilisateur de l'outil qui lui spécifie les chemins d'accès aux bibilothèques de cellules et les formats de fichiers en entrée. En conséquence, les accès disque étant implicitement gérés par le système — à la manière d'une mémoire virtuelle —, le développeur peut considérer qu'il a le droit d'accéder à n'importe quel objet de la structure à tout instant.

7. **accepter divers formats de fichiers.** Les outils développés autour de ce modèle doivent pouvoir s'interfacer avec des chaînes de **CAO** commerciales. Cet interfaçage passe par la possibilité de lire et d'écrire des formats de fichiers standards . Ce service étant assuré sur la structure de données, il soulage les concepteurs d'outils de **CAO** du développement d'analyseurs de grammaire, et donne accès pour chaque outil à un nombre important de formats de fichier. Pouvoir lire de l'**edif**, du **spice**, du **vhdl** structurel, est une conséquence immédiate de l'utilisation de ces structures.

   Le programme utilisateur de la structure n'a pas, par principe, la connaissance du format de fichier utilisé pour la remplir. Ce sont les procédures de lecture et d'écriture qui assurent le respect des sémantiques entre le format externe et la structure de données. Ceci assure un comportement identique des outils quel que soit le format de fichier utilisé pour charger la structure. Accepter un nouveau format de fichier n'aura en principe aucun impact sur les outils existants.

8. **être portable sur de multiples architectures matérielles.** Le développement d'outils de **CAO** et leurs utilisations doivent se faire sur un parc de machines hétérogènes. L'enseignement des méthodes de conception se fait typiquement sur des ordinateurs à bas prix, des **PC**s. Le développement de circuits complexes a lieu sur des stations de travail, type **Sparc** ou **Dec**. L'utilisation d'outils d'**Alliance** à la compagnie **Bull** se

fait sur des machines **Sps9**. Il faut se donner les moyens d'atteindre un large éventail d'utilisateurs.

Ces principes tendent à garantir et à faciliter le développement d'outils de **CAO** communiquant par construction. Ils simplifient considérablement l'interface avec le monde extérieur. L'utilisation d'objets à sémantique rigide a de grandes vertus, cependant il est parfois nécessaire d'enrichir la structure avec des informations supplémentaires. Il existe un champ banalisé partageable dans chaque objet de la structure qui permet cette utilisation.

2.3 La vue physique

2.3.1 Définition de la vue physique symbolique

La vue physique ne vise pas la représentation d'éléments topologiques proche de ceux utilisés pour la fabrication, comme des rectangles ou des polygones. Elle désire au contraire fournir un niveau d'abstraction proche de celui des éditeurs symboliques, en utilisant des objets composites à sémantique plus riche.

Les autres structures de données documentées qui servent à représenter les objets du *VLSI* cherchent également à fournir un certain niveau d'abstraction, mais elles font cela de façon différente.

Les structures de données de **GDT**, la **LDataBase**, reflètent une approche où tous les éléments dérivent d'un objet générique auquel on associe un certain nombre de propriétés également génériques. Ces propriétés vont, par leurs combinaisons, finir par déterminer précisément l'objet. La **LDataBase** est composée de trois structures :

`objbuf`, qui est l'objet générique. Il contient les informations qui sont utilisées par toutes les classes d'objets indifféramment : identificateur, nom, coordonnées,... C'est le champ `class` qui va définir ce qu'est l'objet, et qui sera typiquement rectangle, instance, transistor, connecteur.

`propbuf`, qui est la propriété générique que l'on peut attacher à un objet. Là, c'est l'utilisation combinée de la classe de la propriété — contact par exemple — et de son type — la largeur — qui va définir la sémantique du champ de la structure.

`geombuf`, contient les informations géométriques. C'est la même structure pour tous les objets, et définira typiquement les coordonnées d'un polygone. C'est elle qui contient les informations de connectique.

Dans cette approche, l'interprétation des champs des structures varie dynamiquement.

Les structures proposées par l'université de Berkeley[Cas91] ont une approche similaire. Dans **Octtools**, il y a un objet générique, le `octObject`, duquel dérivent directement tous les autres objets. Cet objet générique contient un identificateur, et une classe qui indique le type de l'objet qui est défini. Les classes ont des champs bien spécifiés, comme par exemple l'instance qui contient, entre autre, un nom de modèle et une opération géométrique. Mais dans ce cas comme dans le cas de **GDT**, le niveau sémantique des classes d'objets n'est pas identique. L'objet générique contient un modèle, ou un couple de coordonnées, ou un niveau technologique, ou un polygone ou encore une équipotentielle.

Les structures de données de CADENCE utilisent une approche différente, plus hiérarchisée, utilisant les techniques d'héritage des langages objets. Il y un objet générique qui contient toutes les classes d'objets, et chaque niveau de la hiérarchie des classes possède l'ensemble des champs qui sont communs à tous les niveaux immédiatement inférieurs. Par exemple, le niveau forme géométrique contient une couche physique et une enveloppe, qui sont applicables à toutes les classes dérivant du niveau, comme polygone , ellipse ou chemin . Chaque classe terminale, c'est à dire qui ne possède pas de niveaux hiérarchiques inférieurs, possède les attributs qui lui sont propres. Dans ce cas, la sémantique des champs est prédéfinie, hormis pour le type de la classe du niveau dérivé.

La propriété de généricité est commune à ces trois approches : si les objets dérivent tous d'une même entité, alors on peut effectuer une opération de manière identique sur tous les objets. C'est ce principe qui est appliqué dans les langages objets, et qui semble avoir été retenu pour ces trois exemples.

Mais généralement on ne fait pas subir à une instance les mêmes traitements qu'à un connecteur ou un segment, car ce sont des objets dont l'utilisation dans la conception est très différente.

L'intérêt d'avoir un objet générique n'est pas clair, car on conçoit difficilement qu'il soit possible de partager une quelconque fonction entre une instance et un niveau technologique par exemple. Le but semble être une complète homogénéisation des paramètres des fonctions d'un point de vue informatique.

Le choix d'identifier *a priori* les différents objets intervenant dans la description du circuit est une démarche plus proche de la vision du concepteur. En ce sens, elle est plus claire et plus pédagogique qu'une approche utilisant des objets génériques. Elle garantie de plus une culture commune aux concepteurs de circuits et aux développeurs d'outils de **CAO**.

2.3.2 Sémantique des éléments de la vue physique symbolique

Ce paragraphe détaille les objets de la vue physique de la structure de données MBK. Les objets nécessaires à la représentation de la vue physique symbolique sont au nombre de six. À chaque object correspond une structure et une seule, et elle possède des champs dont la sémantique est prédéfinie et détaillée dans les paragraphes qui suivent. Le niveau sémantique de chacun des objets est identique : ce sont les objets manipulés par le concepteur sous un éditeur de masques symbolique.

Les descriptions qui suivent définissent un objet en fonction des éléments qui le composent. Ce sont soit des objets — par exemple une figure possède une liste de contacts —, soit des attributs terminaux — un contact possède un couple de coordonnées; ce couple n'est pas un objet dans notre approche —.

Le nombre d'éléments à introduire dans la structure étant inconnu *a priori*, des listes chaînées sont utilisées pour représenter des ensembles d'objets d'un même type.

Figure

Elle constitue l'objet de base d'une description hiérarchique. Une figure est décrite en terme de ses constituants : ceux-ci peuvent être des objets élémentaires ou des instances d'autres figures. Lui sont attachées :

Nom	
Connecteurs	→ liste de connecteurs
Références	→ liste de références
Instances	→ liste d'instances
Segments	→ liste de segments
Contacts	→ liste de contacts
X0, Y0, X1, Y1	
Mode	

- le nom de figure. Ce nom est un identificateur, et doit être unique dans l'espace des noms de figures. Cette unicité est garantie par construction. C'est ce nom qui sera utilisé comme nom de fichier lors de la sauvegarde sur disque.
- un pointeur sur la liste des connecteurs qui définit l'interface de la figure.
- un pointeur sur la liste des références. Une référence associe un nom à une coordonnée (x, y).
- un pointeur sur la liste des instances de figures qui sont utilisées dans la figure courante.
- un pointeur sur la liste des segments, qui représente les fils utilisés dans le dessin. Les transistors sont des segments d'une couche physique particulière.
- un pointeur sur la liste des contacts, qui font la connexion entre deux couches isolées électriquement.
- quatre coordonnées, qui représentent la boîte d'aboutement de la figure. La boîte d'aboutement est différente de l'enveloppe de la cellule puisqu'elle est définie par le concepteur, et est utilisée comme frontière lors du placement côte à côte d'instances. La figure 3.4, page 56, en illustre l'utilité dans le contexte de la génération procédurale.
- le mode de chargement en mémoire. En mode total , toute l'information est chargée en mémoire. Mais dans de nombreux cas, seule l'interface de la figure est utile. En mode partiel , seuls les connecteurs, les segments de transparence et les références sont chargés en mémoire.

L'espace des noms de figures est unique, il ne peut y avoir deux figures de même nom dans la structure. La raison de ce choix est issue des contraintes des formats de représentation hiérarchique des masques réels — **cif** ou **gdsII** —. Le fichier représentant un circuit complet contient tous les modèles instanciés dans la hiérarchie, et si deux modèles différents possédaient le même identificateur, le fichier ne pourrait être généré.

Connecteur

Les connecteurs définissent l'interface d'une figure. Ils indiquent à l'utilisateur de la structure les éléments de la topologie qui correspondent fonctionnellement aux connecteurs logiques de même nom dans la figure logique de même nom. C'est grâce à eux que le lien entre la vue physique et la vue logique s'établit. Ils sont notamment indispensables aux routeurs et extracteurs hiérarchiques. Ils contiennent les informations suivantes :

Nom
Index
Face
X, Y
Largeur
Couche physique

- le nom de connecteur. Contrairement à la plupart des objets de MBK, le nom des connecteurs physiques n'est pas unique. Ceci en application de la règle suivante :

  > *Dans une figure physique, deux connecteurs physiques équipotentiels par l'intérieur possèdent le même nom qui est le nom du connecteur logique qu'ils représentent. Réciproquement, si deux connecteurs physiques possèdent le même nom, ils doivent être équipotentiels.*

  Cette hypothèse est fondamentale pour les outils qui exploitent la hiérarchie, puisqu'elle les informe sur la connectique interne sans entrer dans la figure. Son respect n'est pas garanti par construction, mais doit être garanti par les outils de vérification.

- un index. Cet index est nécessaire pour identifier de façon non ambiguë un connecteur physique, en particulier dans un contexte de description procédurale. En effet, au vu de la règle énoncée ci-dessus, le nom seul n'est pas un identificateur du connecteur. L'index est calculé en fonction de la position du connecteur considéré par rapport aux autres de même nom, et éventuellement de sa couche physique. Le calcul de l'index est fait automatiquement par la fonction permettant d'ajouter un connecteur physique dans la structure. Un exemple est fourni figure 2.1.

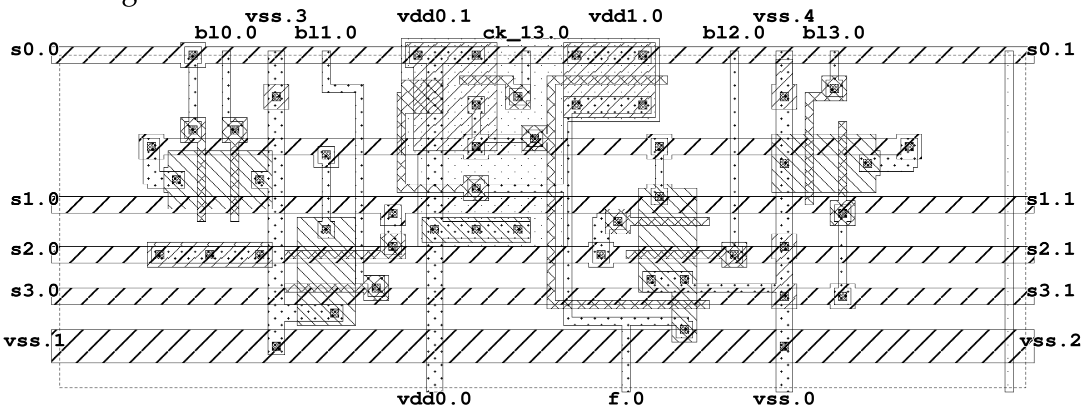


Figure 2.1: Indexation des connecteurs dans la vue physique.

- la face. Elle indique aux routeurs la direction d'accessibilité du connecteur, et peut prendre quatre valeurs, *nord*, *sud*, *est* et *ouest*, imposées par le concepteur. L'appartenance de la face à cet ensemble de valeurs est vérifiée.

- les coordonnées du connecteur. C'est un point situé sur ou dans la boîte d'aboutement de la figure. La présence de connecteurs à l'intérieur de la boîte d'aboutement, bien qu'autorisée, n'est pas souhaitable. De tels connecteurs sont inexploitables par la plupart des routeurs. C'est pour pallier cet inconvénient que la notion de connecteur virtuel , voir **référence** plus loin, a été introduite.

- la largeur. Conceptuellement, le connecteur est vu comme l'extrémité d'un segment. Il possède donc une largeur — généralement égale à la largeur du segment dont il est l'extrémité —. Cette largeur se mesure parallèlement au bord de la boîte d'aboutement. Il possède également une extension dans la direction perpendiculaire au bord de la boîte d'aboutement, mais cette extension dépend de la technologie et n'est donc pas présente dans la structure de données.
- la couche physique. C'est le niveau technologique dans lequel est réalisé le connecteur.

Référence

Les références sont réservées à deux usages précis, qui sont :

1. représentation des connecteurs virtuels. Ils sont généralement placés à l'intérieur des cellules. Les connecteurs virtuels sont exploitables par des routeurs dédiés capables de choisir le point d'accès qui leur convient le mieux parmis un ensemble de points équipotentiels[AG92]. Dans ce cas, le routeur place un *via* sur l'emplacement choisi. Les références n'étant pas matérialisées dans une couche physique, les pistes menant aux références non utilisées peuvent être exploitées pour le routage au dessus des cellules d'autres signaux. L'idée de router au dessus des cellules avec des routeurs choisisant leurs accès devient très intéressante avec les nouvelles technologies, et la notion de référence est prévue à cet effet.
2. manière standard de nommer un point. A titre d'exemple, les références sont utilisées dans la description procédurale du contenu d'une *ROM* ou d'un décodeur de *RAM*.

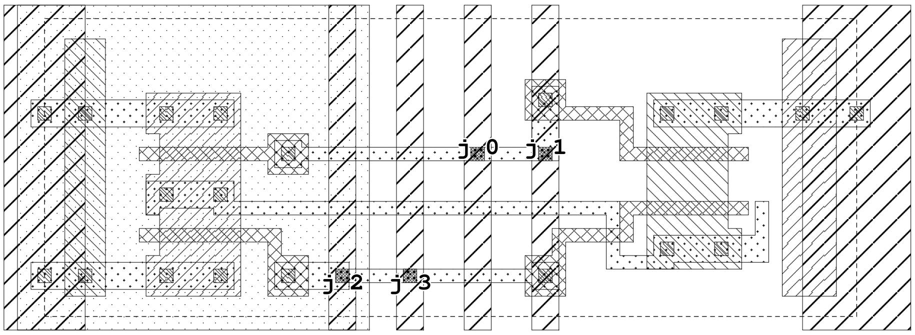


Figure 2.2: Utilisation des références dans un décodeur. Les références notées `ji` sont remplacées par des contacts dans les divers cas possibles du décodage.

Elles sont constituées de :

Nom
Type

- un nom. Il n'y a aucune garantie sur l'unicité du nom.

- un type. Il permet de distinguer les connecteurs virtuels des points nommés. Par défaut deux types sont définis, `ref_con` pour les connecteurs virtuels et `ref_ref` pour les points nommés, mais la fonction d'ajout permet de spécifier n'importe quel autre type.

Instance

C'est l'appel d'une figure dans un contexte particulier. Les informations pertinentes sont :

Nom
Nom de modèle
X, Y
Symétrie

- le nom de l'instance. C'est l'identificateur de l'instance, et doit donc être unique pour une figure donnée. Cette unicité est garantie par construction.
- le nom du modèle qui est instancié. Utiliser le nom du modèle plutôt qu'un pointeur sur la figure instanciée permet d'instancier sans charger le modèle en mémoire. Ceci évite d'avoir à charger récursivement l'ensemble des modèles utilisés dans une figure jusqu'aux éléments terminaux, ce qui est rarement nécessaire. Ceci limite à la fois l'occupation mémoire et le temps de chargement d'une figure.
- les coordonnées du point d'appel de l'instance. Ces coordonnées correspondent au coin bas gauche de la boîte d'aboutement de la figure instanciée. Ce point est invariant par opération géométrique, ce qui signifie que les symétries et rotations se font en place, et que le point d'appel est considéré après que l'opération ait été effectuée. Ce choix est différent de celui des représentations **cif** et **gdsII** pour lesquels l'opération géométrique est effectuée par rapport au repère de définition du modèle de l'instance — ce qui s'explique par l'absence de la notion de boîte d'aboutement dans ces formats—. L'invariance de la coordonnée d'instanciation par opération géométrique est une propriété très intéressante pour la génération procédurale.
- l'opération géométrique éventuelle. L'ensemble des opérations géométriques est constitué de huit éléments constituant un groupe par l'opération de composition. À chaque instance n'est associée qu'une unique opération géométrique. On vérifie lors de la création de l'instance que l'opération géométrique appartient bien au groupe. L'opérateur de composition n'est pas commutatif. Leur sémantique est décrite dans la figure 2.3.
  Il faut noter que pour des raisons de compacité ni la boîte d'aboutement, ni les connecteurs de l'instance ne sont explicitement représentés dans la structure de données MBK.

Segment

La principale utilisation d'un segment correspond à la réalisation physique d'un fil d'interconnexion. Cependant, la description d'un transistor nécessite exactement les mêmes informations que la description d'un fil. Par conséquent, les transistors sont représentés comme des segments d'une couche physique particulière. Les segments sont formés de :

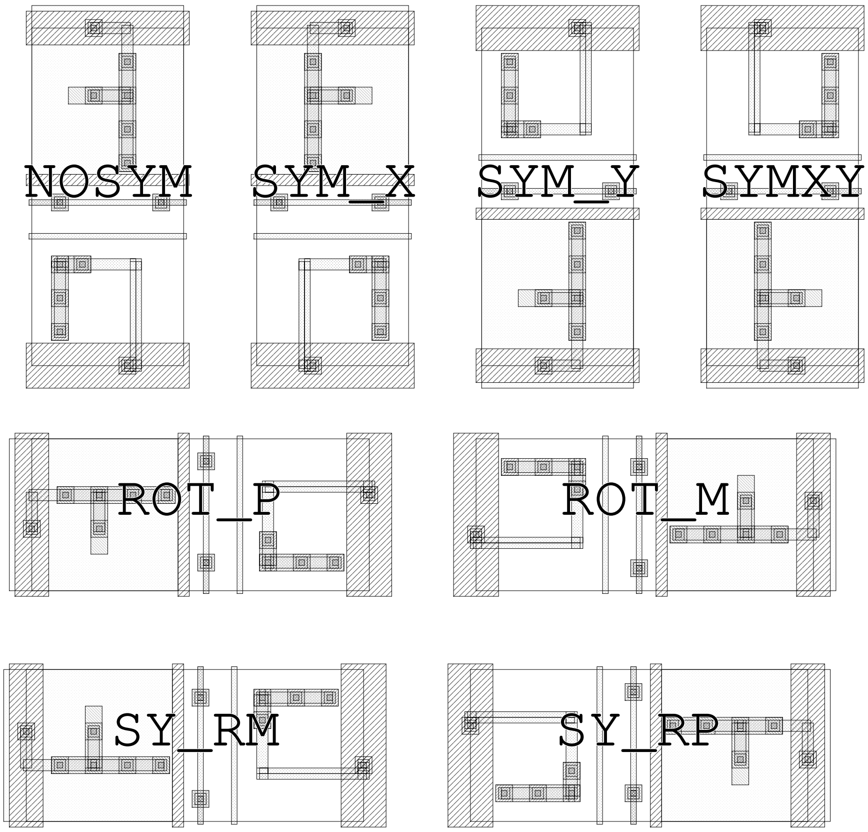


Figure 2.3: Sémantique des opérations géométriques. Le point d'appel de l'instance est dans tous les cas le point bas gauche.

Nom
X0, Y0, X1, Y1
Type
Largeur
Couche physique

- un nom. Ce n'est pas un identificateur du segment, mais un commentaire. Il est optionnel.
- deux couples de coordonnées. C'est un bipoint, tel que, si *(x0, y0)* et *(x1, y1)* en sont les extrémités, alors on aura toujours *x0 = x1* ou *y0 = y1*. Autrement dit un segment est toujours vertical ou horizontal. Le respect de cette contrainte est vérifié par construction, et l'on assure également dans la structure de données que *y0* $\leq$ *y1* pour les segments verticaux et que *x0* $\leq$ *x1* pour les segments horizontaux.
- un type. C'est une information précisant l'orientation du segment, indiquée par l'utilisateur qui peut prendre les valeurs *gauche*, *droite*, *haut* et *bas*. Le type est particulièrement utile lors de la transformation en rectangles des transistors qui ne sont pas nécessairement symétriques par rapport à l'axe du segment, comme

les transistors déplétés en technologie **AsGa** ou **NMOS**.

- une largeur. C'est l'extension en largeur du fil perpendiculairement à son axe de direction. De même que pour les connecteurs, l'extension longitudinale dépend de la technologie et n'est donc pas présente dans la structure de données.
- une couche technologique.

Les transistors sont des segments d'une couche particulière, et sont macro-générés lors de la transformation vers la technologie cible.

Contact

Ils relient électriquement deux couches physiques qui ne sont pas équipotentielles par construction. Ils sont représentés par :

X, Y
Type

- un point. C'est le centre du contact.
- un type qui indique les couches misent au même potentiel par la présence du contact.

La réalisation d'un transistor coudé — forme en `L` — utilise deux segments et un contact, comme illustré figure 2.4.

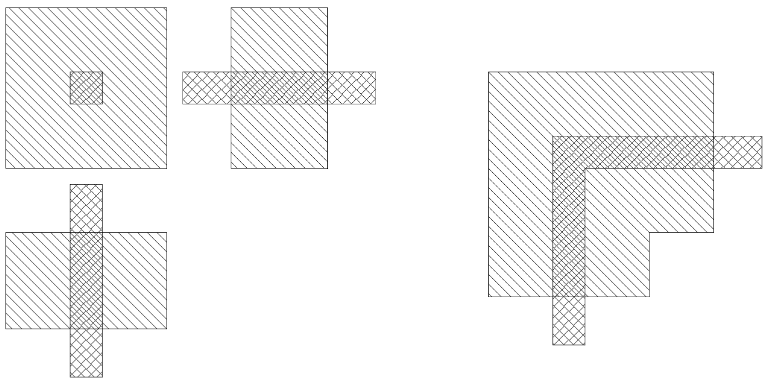


Figure 2.4: Construction d'un transistor coudé.

La figure 2.5 présente, sous une forme épurée, l'organisation de la structure de données pour la figure contenant deux connecteurs, deux segments, un contact et une boîte d'aboutement.

2.3.3 Conclusion

Cette structure est compatible avec l'approche symbolique définie au chapitre 1. Elle permet donc de représenter une description topologique indépendante de la technologie. Sa structuration par objets et les choix sur la sémantique de leurs attributs la rendent bien adaptée aux besoins de la génération procédurale.

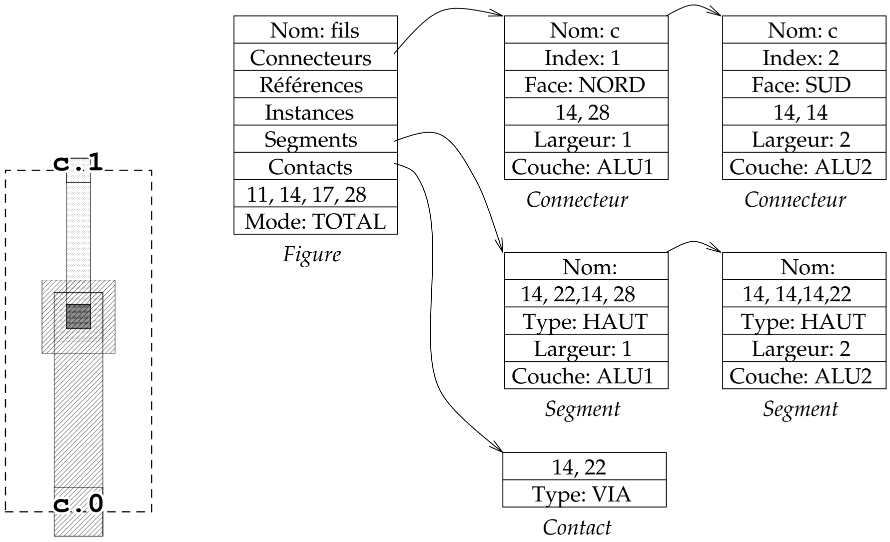


Figure 2.5: Exemple de structure physique

Cette représentation de la vue physique est constituée d'objets beaucoup plus typés qu'une structure de données rectangles ou polygones correspondant aux formats de fichiers **cif** ou **gdsII**. Ceci explique le petit nombre de formats de fichiers externes pour lesquels existent des fonctions de lecture/écriture sur disque :

1. le format **cp** est un format interne au système COMPASS[Kin85]. Ce format permet d'utiliser la fenêtre `compose` comme éditeur de *layout* symbolique, et donc l'interfaçage avec COMPASS.

2. le format **mg** développé chez PHILIPS a été retenu par le projet ESPRIT **IDPS** comme standard d'échange entre les partenaires pour le dessin des cellules de base utilisées dans les générateurs. Il permet d'entrer dans le système MODGEN qui comporte un éditeur graphique, `grapmg`, et un compacteur.

3. le format **ap** est le format interne de la chaîne **Alliance**. Ce format permet également l'interfaçage avec le système CADENCE grâce à l'écriture en langage **Skill** de fonctions de lecture/écriture vers les structures de données internes de CADENCE.

2.4 La vue structurelle

2.4.1 Définition de la vue structurelle

La vue structurelle se veut l'intersection des notions contenues dans les langages standards de représentation de *netlist*. C'est en effet l'étude comparée de la vue **edif**[EDI87] *netlist*, de **vhdl**[SW85] structurel, et du format **hnl**[Shu85] de la chaîne COMPASS, qui a conduit à la définition de la structure MBK logique. Par la suite, nous avons également pris en compte les contraintes liées au format **spice**[Nag75].

Une des retombées les plus intéressantes de cette approche est de fournir un traducteur universel permettant de convertir aisément un format dans un autre.

Cette vue permet de représenter un niveau hiérarchique d'une description structurelle hiérarchique au niveau transistors, portes ou blocs fonctionnels.

Le principal problème lié à cette formalisation est l'identification des objets et des attributs canoniques de la vue structurelle, décrite au paragraphe 2.4.2. L'autre aspect complexe est liée à la gestion de la hiérarchie. Certains formats, **spice** et **edif** par exemple, représentent une figure sous la forme d'un arbre contenant l'intégralité de la hiérarchie. Un seul fichier est nécessaire pour décrire une figure, ses fils, et ainsi récursivement jusqu'aux éléments terminaux. Ces fichiers sont autosuffisants, et garantissent lors de leur lecture la cohérence entre un modèle et ses instances. D'autres formats, comme **vhdl** et **hnl**, permettent de décrire un niveau hiérarchique par fichier. Ainsi, un circuit est décrit par un ensemble de fichiers, chacun de ces fichiers décrivant un père en fonction de ses fils directs. Du point de vue des outils de **CAO**, il n'est pas utile d'avoir *a priori* la totalité de l'arborescence en mémoire. Au contraire, il est très possible dans une conception descendante d'avoir la spécification de l'interface d'un bloc sans en connaître le contenu. On choisi donc dans la structure de représenter un père en fonction de ses fils directs. Ainsi, le chargement des fichiers type **vhdl** ou **hns** ne pose pas de problèmes. Mais pour les fichiers de type **spice** ou **edif**, il faut adopter une stratégie qui permette d'avoir toute l'arborescence en mémoire : la stratégie consiste à charger la totalité du fichier, c'est-à-dire tous les modèles instanciés, en mémoire.

2.4.2 Sémantique des éléments de la vue structurelle

Les objets nécessaires à la représentation de la vue structurelle sont au nombre de cinq. De même que pour la vue physique symbolique, à chaque objet correspond une structure et une seule, qui possède des champs dont la sémantique est prédéfinie. Le niveau sémantique de chacun des objets est identique : ce sont les objets manipulés par le concepteur sous un éditeur de schémas.

Les descriptions qui suivent définissent un objet en fonction des éléments qui le composent. Ce sont soit des objets — par exemple une figure possède un ensemble de connecteurs —, soit des attributs terminaux — un signal possède une capacité —.

Le nombre d'éléments à introduire dans la structure étant inconnu *a priori*, des listes chaînées sont utilisées pour représenter des ensembles d'objets d'un même type.

Figure

Elle décrit un bloc père en terme de ses blocs fils directs, les instances, interconnectés entre eux au moyen d'équipotentielles reliant les connecteurs des fils et les connecteurs du père. Elle est constituée de :

Nom	
Connecteurs	→ liste de connecteurs
Instances	→ liste d'instances
Transistors	→ liste de transistors
Signaux	→ liste de signaux
Mode	

- un nom. C'est l'identificateur de la figure, et il doit être unique dans l'espace des noms de figures. Cette unicité est garantie par construction. De même que pour la vue physique, c'est ce nom qui sera utilisé comme nom de fichier lors de la sauvegarde sur disque.

- un pointeur sur une liste de connecteurs. Ils définissent son interface logique. Ces connecteurs sont dits *externes*, par opposition aux connecteurs des instances qui sont dits *internes*.
- un pointeur sur une liste d'instances. Ce sont les références aux modèles utilisés dans le père.
- un pointeur sur une liste de signaux. Ce sont les équipotentielles qui représentent les connexions entre les objets constitutifs de la cellule.
- un pointeur sur une liste de transistors.
- le mode de chargement. En mode total , toute l'information est chargée en mémoire. Mais dans de nombreux cas, seule l'interface de la figure est utile. En mode partiel , seuls les connecteurs et les signaux qui y sont attachés sont chargés en mémoire.

La vue logique et la vue physique d'une même figure possèdent le même nom, qui est également le radical de leurs noms de fichier. Réciproquement, si le nom d'une figure logique est identique au nom d'une figure physique, alors ce sont les deux vues du même objet.

Connecteur

Spécifiant l'interface d'une figure logique, les connecteurs contiennent :

Nom	
Signal	→ signal unique
Type	
Racine	→ Figure, instance ou transistor
Direction	

- un nom, qui est leur identificateur dans la figure. La correspondance entre l'interface des vues physique et logique est illustrée ci-après, figure 2.6. Cette unicité est garantie par construction. Dans la vue physique et la vue logique d'une même figure, la correspondance se fait par nom.
- un signal. Il représente l'équipotentielle à laquelle le connecteur est relié. Par définition, deux connecteurs d'une figure ne peuvent être connectés au même signal, car ils seraient équipotentiels par l'intérieur. Cette propriété est garantie par construction.
- un type, qui indique si le connecteur appartient à une figure, à une instance, ou à un transistor. En l'occurrence, pour une figure le type sera toujours *externe*, mais l'objet connecteur est également utilisé dans les instances, il est alors de type *interne*, ou dans les transistors, il est alors de type *transistor*.
- un pointeur sur l'objet auquel il appartient — figure, instance ou transistor —.
- une direction, qui indique la manière dont l'information se propage à travers le connecteur. Les directions reconnues sont détaillées dans la figure 2.7.

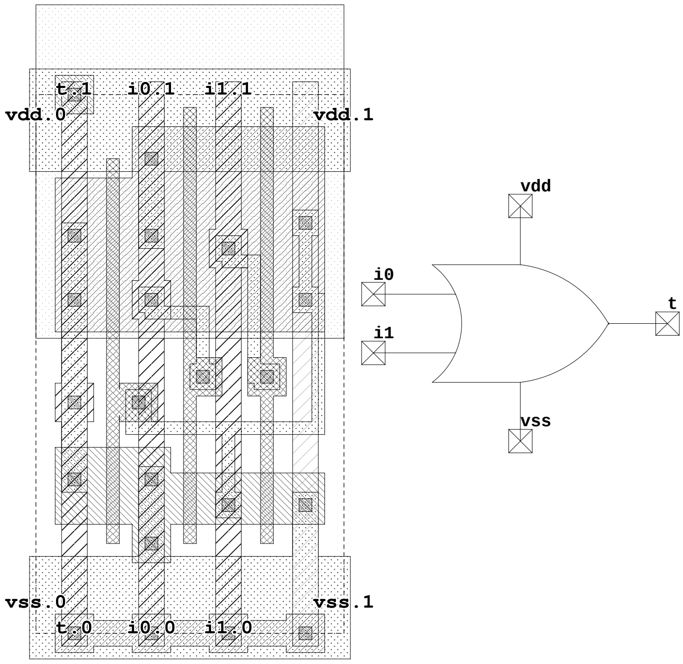


Figure 2.6: Correspondance entre connecteurs physiques et logiques.

On constate que la connectique est décrite du point de vue des connecteurs : chaque connecteur connaît le signal auquel il est connecté. Une fonction utilitaire permet de construire la représentation duale — on attache à chaque signal la liste des connecteurs qui lui sont connectés — en utilisant le champ banalisé de la structure de données.

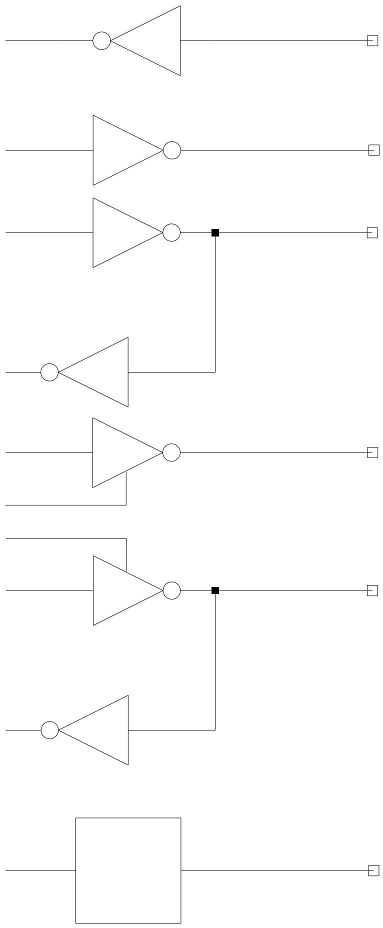
La structure de données MBK fournit certaines garanties de cohérence. Cependant, ces garanties ont des limites : il existe des règles de validité électriques sur un signal qui dépendent de la direction de l'ensemble des connecteurs qui y sont attachés. Ces règles ne sont pas vérifiées, et l'on peut donc représenter des *netlists* électriquement incorrectes dans la structure de données.

Instance

L'instanciation est l'appel d'un modèle dans un contexte particulier. Les informations nécessaires pour cette opération sont :

Nom	
Nom de modèle	
Connecteurs	→ liste de connecteurs

- le nom d'instance, qui est l'identificateur de l'instance dans la figure.
- un nom de figure, qui est le nom du modèle que l'on instancie. Utiliser le nom du modèle plutôt qu'un pointeur sur l'objet permet d'instancier sans charger le modèle en mémoire effectivement. Ceci évite d'avoir à charger récursivement l'ensemble des modèles utilisés dans une figure jusqu'aux éléments terminaux.
- une liste de connecteurs. C'est une liste qui contient autant de connecteurs qu'en possède la figure qui est instanciée. Un connecteur d'instance est dit de type



Le connecteur est une entrée, (`IN`).

Le connecteur est une sortie, (`OUT`).

Le connecteur est une sortie, mais le signal qui en sort est réutilisé en interne, (`INOUT`).

Le connecteur est une sortie, avec posibilité de mise en haute impédance, (`TRISTATE`).

Le connecteur est une sortie, avec posibilité de mise en haute impédance, et le signal qui en est issu est réutilisé en interne, (`TRANSCV`).

La direction du connecteur est inconnue, (`UNKNOWN`).

Figure 2.7: Directions possibles des connecteurs logiques. Ce typage respecte la sémantique **vhdl**.

interne. La structure de données garantit que l'ordre des connecteurs dans la liste de l'instance d'un modèle est identique à l'ordre des connecteurs dans le modèle instancié.

Transistor

C'est l'élément terminal de toute figure si l'on met à plat la hiérarchie. Il est constitué de :

Type	
Drain	→ connecteur unique
Grille	→ connecteur unique
Source	→ connecteur unique
Largeur, longueur	
X, Y	
AS, PS, AD, PD	

- un type. Les technologies visées par la structure de données étant le **MOS** et l'**AsGa**, deux types de transistors sont actuellement définis. Ils permettent de différencier les transistors N et P en **CMOS** et les transistors enrichis et déplétés en **AsGa**.

- trois connecteurs, représentant les grille, drain et source du transistor. Ces connecteurs sont de type *transistor*. La polarisation du substrat est implicite.
- la largeur et la longueur de grille.
- un couple de coordonnées qui représente la position du transistor.
- les périmètres et surface de drain et de source.

Les informations de position, de périmètre et d'aire du drain et de la source ne sont documentées que lors de la manipulation de *netlist*s extraites.

Signal

Le signal relie électriquement un ensemble, éventuellement vide, de connecteurs. Un signal est, au plus, relié à un connecteur externe. En effet, si deux connecteurs d'une figure étaient équipotentiels, ils porteraient le même nom, ce qui n'a pas de sens pour la vue logique. Pour le définir, sont utiles :

Index	
Noms	→ liste de noms
Type	
Capacité	

- un index, qui est l'identificateur du signal. Deux signaux ne peuvent posséder le même index dans une figure donnée.
- une liste de noms. Cette liste contient les noms synonymes du signal. Ces noms sont fournis par l'utilisateur lors de la conception, et la liste peut donc être vide. Un signal peut avoir plusieurs nom à la suite d'une opération de mise à plat de la hiérarchie. Néanmoins il est garanti que deux signaux ne peuvent avoir un synonyme commun.
- un type, qui indique si le signal est connecté à un connecteur externe, ou s'il n'est relié qu'à des connecteurs d'instances ou de transistors.
- une capacité, qui est la somme des capacités ramenées au substrat de tous les fils sur l'équipotentielle, exception faite des diffusions et grilles de transistors.

Les informations de capacités ne sont documentées que pour les *netlist*s extraites.

On notera que la description d'une figure nécessite la connaissance de l'interface des divers modèles instanciés, mais non de leur contenu.

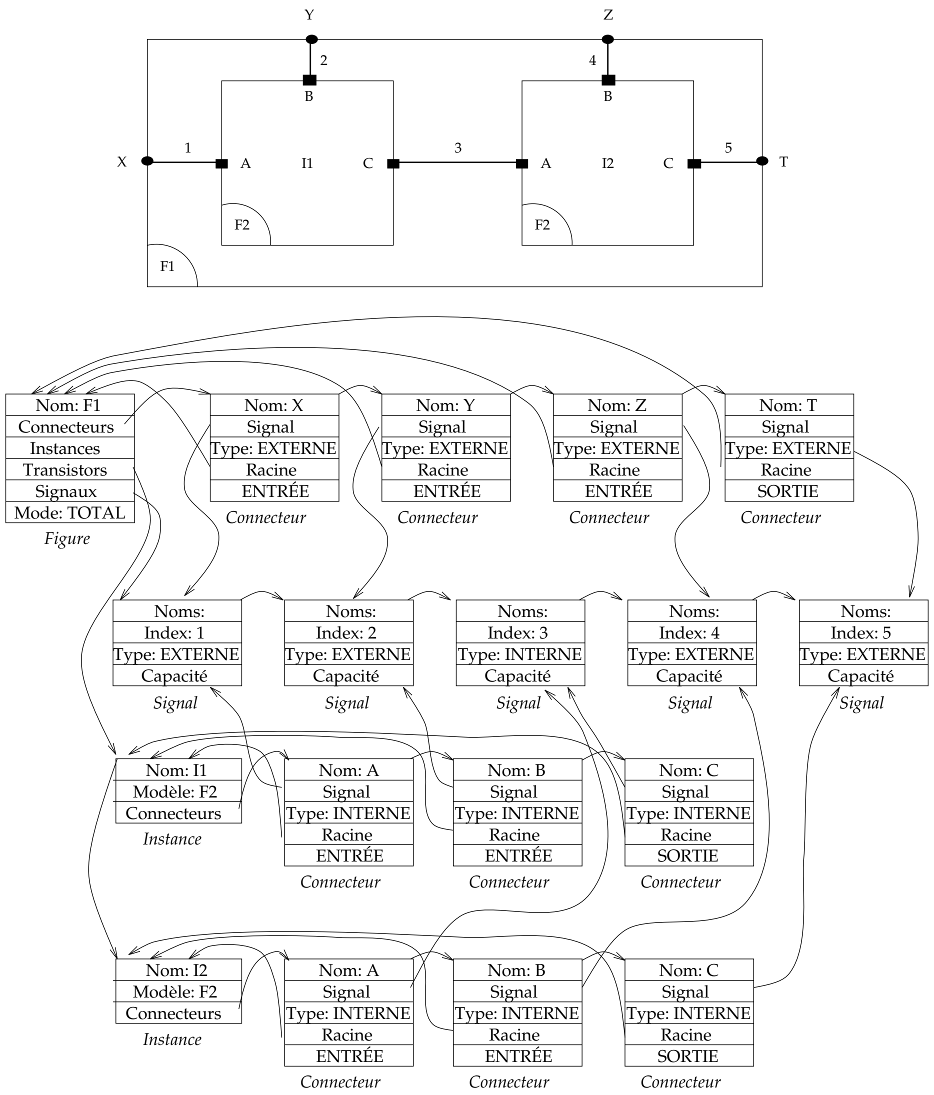


Figure 2.8: Exemple de structure logique

La figure 2.8 présente, sous une forme symbolique simplifiée, l'organisation de la structure de données pour une figure instanciant deux fois le même modèle. La figure est chargée en mode total . On notera l'absence de capacité sur les fils, car cette représentation n'est ni extraite des masques, ni annotée par son concepteur.

2.4.3 Conclusion

La vue logique a été définie à partir de l'étude de formats de fichier standards. Sont actuellement supportés :

1. le format **vst**, le sous ensemble structurel du langage **vhdl**. Ce format ne sait représenter ni les transistors, ni les capacités.

2. le format **spice**. Il est la cible privilegiée des *netlist*s extraites. Il permet l'utilisation d'outils de simulation électrique ou d'analyse temporelle.

3. les formats **hnl** et assimilés, qui sont nécessaire à l'utilisation des outils COMPASS.
4. le format **al**, qui est le format interne de la chaîne **Alliance**. Il est une image textuelle fidèle du contenu de la structure de données.
5. le format **edif**, qui permet l'interfaçage avec les outils de CADENCE.
6. le format **cct**, destiné à fournir des *netlist*s en entrée de l'outil `hilo` de GENRAD. Ceci permet d'avoir accès aux outils de test.

2.5 Implémentation

Les objets MBK et les fonctions de bas niveau qui les manipulent sont décrits en langage **C** pour des raisons :

- de performances, car le **C** est un langage compilé performant.
- de portabilité, car ce langage est disponible sur de nombreuses machines, étant fourni avec la plupart des systèmes UNIX. De plus, c'est un langage qui est enseigné dans toutes les universités, clairement documenté par un comité de standardisation, et qui possède un environnement de programmation performant.
- de logistiques, liées à la possibilité d'archiver les fonctions sous forme de librairies aisément partageables.

L'implémentation en **C** correspond à la mise en œuvre du principe visant à garantir un large éventail d'architectures matérielles cibles.

2.5.1 Accès aux objets

Pour chacun des objets MBK, il est défini un ensemble de procédures standards d'accès.

Encapsulation :

Ce que désire voir le développeur d'applications est la sémantique des objets qu'il manipule et non leur implémentation. La structure de données et son organisation doivent être relativement cachées, afin que lors de modifications, soit pour améliorer les performances, soit pour ajouter de l'information à un objet, il n'y ait pas de répercussion sur le code existant, mais simplement mise à disposition d'une nouvelle fonction. Ceci n'est pas particulièrement nouveau, et s'inscrit dans une approche orientée objet avec un langage procédural classique. La maintenance et l'évolution du code des algorithmes ne doit pas dépendre des évolutions de la structure de données.

D'autre part, cette encapsulation donne au développeur l'illusion que toutes les données qu'il manipule sont présentes en mémoire. Cela permet de ne charger du disque que les informations pertinentes à un instant donné. Si l'utilisateur désire accéder à une information qui n'est pas dans la structure, alors la fonction d'accès la chargera à partir du disque.

Cohérence :
Il faut garantir un certain niveau de cohérence interne de la structure de données. Le programmeur ne doit pas venir forcer la valeur d'un champ arbitrairement, car il risquerait de perturber la cohérence de la structure. Prenons le cas des connecteurs de la vue physique. Comme l'on désire être capable de désigner un connecteur de manière non ambiguë, et que deux connecteurs équipotentiels par l'intérieur d'une *figure* physique doivent porter le même nom, seules leurs coordonnées et la couche physique qui les composent peuvent permettre de les différencier. Afin d'avoir, une méthode générale permettant l'identification, un index est calculé et attaché au connecteur. Ainsi, si l'on va modifier directement les coordonnées, la couche physique, ou l'index du connecteur, la structure de données est rendue incohérente.

Les fonctions d'accès à la structure de données sont de trois types :

Ajout :
permet de créer l'objet, avec les attributs fournis par l'utilisateur lors de l'appel. Ces fonctions garantissent la cohérence de la structure afin qu'elle satisfasse aux définitions des paragraphes 2.3.2 et 2.4.2.

Recherche :
permet, connaissant l'identificateur de l'objet — en général son nom — de récupérer un pointeur sur l'objet.

Destruction :
detruit physiquement l'objet désigné par son pointeur.

Tous les éléments de la structure de données sont stockés dans des listes simplement chaînées. L'ordre des objets dans une liste est généralement non significatif, et l'insertion d'un objet dans une liste se fait en tête. Tout objet possède les deux champs suivant :

`NEXT`
c'est le pointeur de chaînage sur l'élément suivant. Il est mis à `NULL` par convention sur le dernier élément de la liste. Afin de permettre d'utiliser des fonctions génériques sur les listes de pointeurs d'objets MBK, comme la fonction `reverse` qui inverse le chaînage, ce champ se trouve déclaré au début de tous les objets. Ceci garantit la portablilite des fonctions génériques, comme cela est spécifié dans la norme du **C**.

`USER`
ce champ est mis à disposition des programmes qui désirent attacher des informations supplémentaires à un objet. Une procédure d'accès est définie afin d'éviter les collisions. À la différence de l'*héritage* des langages à objets, il est possible à plusieurs programmes utilisateurs partageant la même structure de données d'ajouter indépendamment de l'information sans conflit.

Afin d'éviter le problème de la fragmentation de la mémoire, qui est crucial dès que l'on occupe beaucoup de mémoire en allouant des blocs de petite taille, des techniques d'allocations par blocs ont été mises en œuvre pour les structures les plus utilisées.

2.5.2 Gestion des noms

Nombreuses sont les structures de MBK qui possèdent un champ `NAME`. Il est soit un identificateur de l'élément, soit une information destinée à l'utilisateur dans le cas où l'identificateur n'est pas parlant. Les deux points importants concernant l'usage qui doit être fait des noms sont :

Dictionnaire
: afin d'accélérer les recherches par noms, il existe un dictionnaire dans lequel chaque nom introduit dans la structure de données par les fonctions d'accès est archivé de manière unique. Les noms des objets étant représentés par des pointeurs vers le dictionnaire, on peut ensuite, lors de parcours sur MBK, tester l'égalité des pointeurs au lieu de tester l'égalité des chaînes de caractères. En contrepartie, si un nom est introduit illégalement dans la structure, ou qu'un nom déjà introduit est modifié, alors la structure est rendue inutilisable.

Représentation des vecteurs
: certains formats de représentation structurelle, comme **edif** ou **vhdl**, autorisent explicitement la manipulation de vecteurs comme entité, pour les connecteurs et les signaux. Cette notion n'existe pas en tant que telle sur les objets MBK, car elle n'ajoute pas d'information utile sur la *liste d'interconnexion* décrite. C'est en quelque sorte une facilité mises à disposition de l'utilisateur dans la description source. Afin de ne pas surcharger la sémantique de la structure de données, et de ne pas pénaliser les formats de bas niveau, tel **spice**, la solution adoptée consiste à coder l'appartenance d'un objet à un vecteur dans son nom. On sait donc représenter un élément de vecteur, charge étant aux outils désirant exploiter l'entité vecteur d'en retrouver tous les membres. Cependant, comme le groupage des connecteurs vectorisé est important dans certains formats de fichiers, une convention stipule que les éléments d'un vecteur se suivent dans la liste, et la variation de l'index est monotone croissant ou décroissant de pas un. La convention de codage est `nom index`, ou `nom` est le nom commun aux éléments du vecteur, et `index` son numéro.

2.5.3 Architecture en librairies

Les fonctionnalités mise à la disposition des outils utilisant les objets MBK sont de quatre niveaux :

1. **fonctions utilitaires bas niveau**. Ce sont des fonctions génériques qui agissent sur des structures permettant la représentation de listes ou de graphes simples. Les fonctions de bas niveau sont utilisées par les fonctions d'accès aux objets MBK, mais elles peuvent aussi être utilisées dans un contexte différent. Des fonctions de gestion des tables de hachages, ainsi que de manipulation des chaînes de caractères sont également fournies et documentées.

2. **fonctions d'accès aux objets des vues structurelle et physique symbolique**. Ce sont les fonctions de base, ajout, recherche, destruction. Elles emploient les fonctions utilitaires bas niveau.

3. **fonctions de lecture et d'écriture des formats de fichiers connus de** MBK. Il existe deux fonctions par format de fichier : lecture et écriture. Ces fonctions ont besoin des fonctions d'accès aux objets et des fonctions utilitaires.

4. **fonctions utilitaires de haut niveau**. Elles fournissent un ensemble de fonctionnalités de haut niveau qui sont utilisées lors de l'implémentation d'algorithmes complexes. On y trouve par exemple les fonctions de mise à plat de la hiérarchie. Elles utilisent des fonctions des trois niveaux inférieurs.

   Ce découpage hiérarchique a trois avantages :

- il permet de donner une vision claire de la différence de niveau sémantique des fonctions disponibles aux développeurs d'outils de **CAO**.
- il garantit l'absence de cycles dans le graphe des dépendances entre les librairies : aucune fonction d'une librairie de bas niveau n'appelle de fonction d'une librairie de plus haut niveau. Ainsi, l'édition de liens lors de la compilation des applications se passe bien sur tous les systèmes.
- il facilite la maintenance des fonctions de lecture et d'écriture de fichiers. En effet, ce type de fonctions est sujet à erreurs, car différentes sources produisent différents jargons pour un même langage standard . On peut ainsi assurer l'évolution d'une partie des librairies sans interférer sur d'autres tant que les interfaces sont stables.

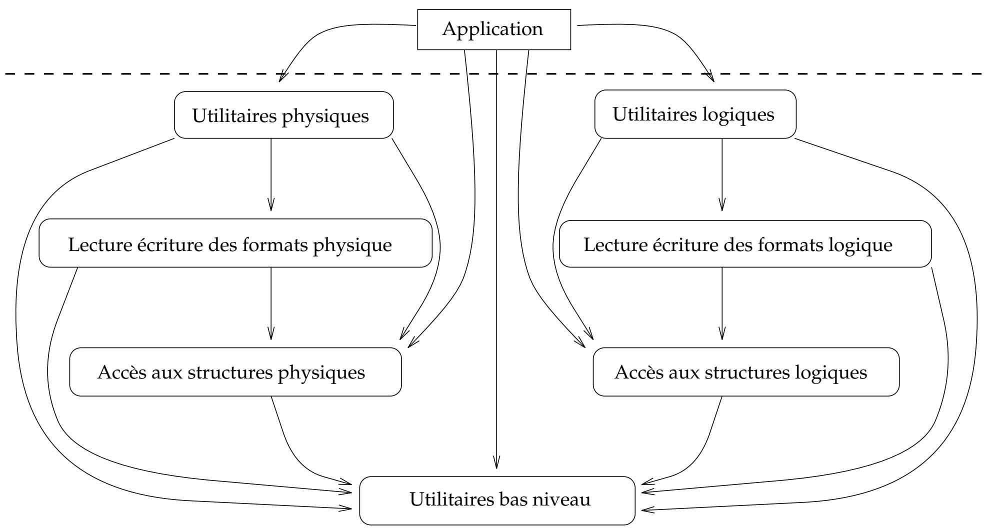


Figure 2.9: Structuration hiérarchique des librairies MBK.

La figure 2.9 illustre les dépendances entre les applications et les librairies MBK. Elle mérite quelques commentaires.

- le symbole $A \leftarrow B$ indique que A dépend de B, c'est à dire qu'il en utilise les objets et les fonctions. Notons que la dépendance dont il est question ici est une dépendance vis-à-vis des interfaces et de l'action des fonctions. Ainsi, une fonction de B peut être modifiée du point de vue de son implémentation, afin de la rendre plus performante par exemple, mais ne nécessitera pas pour autant une mise à jour de A puisque ni son interface, ni sa sémantique n'ont été changées.

- la vue physique et la vue logique sont clairement dissociées. Un outil ne travaillant que sur la vue structurelle ne sera compilé qu'avec les librairies de la partie droite du dessin.
- les fonctions de lecture et d'écriture ne sont pas accessibles directement par les applications. C'est la couche fonctions utilitaires logiques et physique qui se charge de vérifier qu'une figure est en mémoire lorsque l'application la demande, et de la charger du disque si elle n'y est pas.

2.6 Résultats

Depuis quatre ans, une trentaine de chercheurs en **CAO** et une quinzaine d'étudiants par an ont écrit environ 300.000 lignes de **C** autour de ces structures de données. Ceci illustre la qualité de la sémantique fixe, qui, si elle rigidifie les interfaces et fait perdre en souplesse pour le programmeur, garantit en contrepartie le respect des sémantiques et facilite la communication entre applications.

L'aspect pédagogique de l'approche n'est pas négligeable non plus. Ces structures ont été utilisées par des programmeurs inexpérimentés sur des laps de temps très courts — de l'ordre de deux à trois semaines réparties sur quatre mois —, pour le développement d'outils simples ou de fonctions d'outils complexes.

Les outils de la chaîne **Alliance** travaillant sur la vue logique ou physique exploitent tous les structures de données MBK. Cette chaîne est constituée d'une trentaine d'outils,

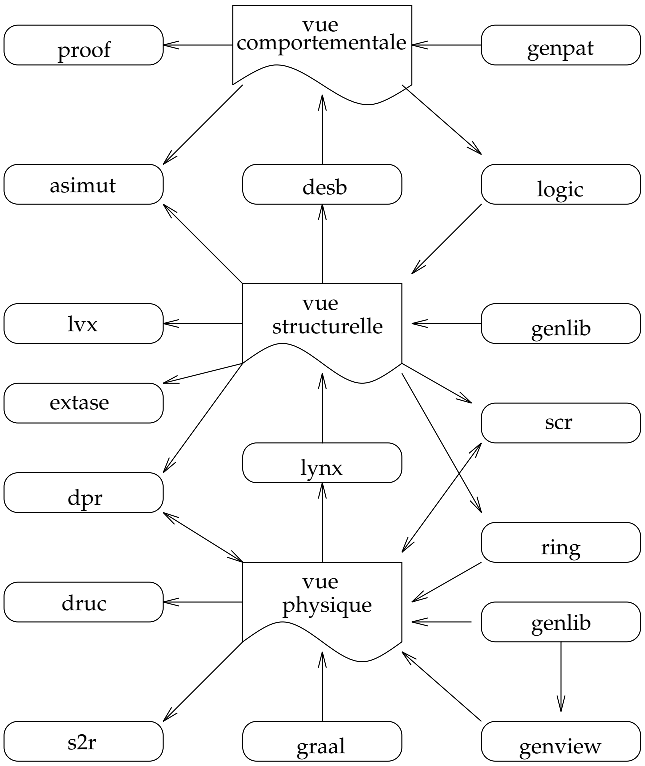


Figure 2.10: Interfaçage des principaux outils d'**Alliance** avec les structures MBK.

basés sur une vingtaine de bibliothèques de fonctions. Elle a été développée en quatre ans, le codage des outils se faisant en parallèle car leur interdépendance est limitée au respect des sémantiques attachées aux objets de la structure de données MBK. La figure 2.10 illustre cet interfaçage — la vue comportementale qui apparaît ici n'a pas été définie dans le cadre de ce travail —.

les outils d'**Alliance** peuvent être utilisés pour pallier l'inexistence ou l'inefficacité d'un outil d'une chaîne de **CAO** commerciale, car la structure de données a été conçue pour l'interopérabilité. Grâce à cela, ils s'intègrent dans des méthodologies de conception variées.

Par exemple, la compagnie BULL utilise les éditeurs de masques et de schémas de COMPASS, mais n'est pas satisfaite du langage de génération procédurale de cette chaîne. Le langage GENLIB présenté au chapitre 3, a été adopté entre autre pour la réalisation physique et le modèle structurel d'un cache de 800.000 transistors utilisant les cellules saisies sous COMPASS.

L'outil `tas`[HMGK91], l'analyseur temporel statique de la chaîne **Alliance**, a été utilisé chez SGS-THOMSOM et prend en entrée une liste d'interconnexions de transistors et de capacités de type MBK.

L'ensemble des sources des bibliothèques des vues physique et structurelle, non incluses les fonctions de lecture/écriture de fichiers, fait environ 9000 lignes de **C**.

2.7 Conclusion

Nous avons présenté dans ce chapitre des structures de données qui fournissent une représentation unifiée des objets de la conception *VLSI*, pour les vues physique et structurelle. Il est difficile de prouver dans l'absolu et de façon objective la supériorité d'une structure de données de ce type — qui n'est pas dédiée à un algorithme particulier — par rapport à une autre. On peut cependant juger *a posteriori* de la pertinence des choix de principe:

- utiliser des objets à sémantique rigide permet d'obtenir des outils qui communiquent par construction avec les autres outils construits sur la structure. La chaîne **Alliance** est constituée de nombreux outils, et ils communiquent tous entre eux sans nécessiter l'usage de traducteurs multiples. Ajoutons que la technique des traducteurs ne permet pas toujours de réduire les incompatibilités sémantiques entre outils.

  La correspondance un pour un entre les objets et leur implémentation informatique permet d'avoir une vision simple et bien connue pour le développement d'outils. L'ampleur du travail fait autour de ces structures en peu de temps prouve que ce modèle est facile à appréhender.

- séparer la vue physique de la vue logique. Ce choix, qui peut paraître discutable à première vue, s'avère judicieux :
  les outils qui ne manipulent qu'une vue ne sont pas pénalisés par une consommation mémoire inutile. En réalité, seuls un nombre restreint d'outils ont besoin des deux vues;
  le coût, en temps d'exécution, qu'impliquerait une vérification de conformité entre les deux vues lors de chaque ajout ou effacement est prohibitif;

la cohérence entre les vues physique et structurelle ne peut pas être assurée implicitement par la structure, car cette notion est très dépendante des applications. C'est donc aux outils qui exploitent les deux vues d'assurer cette cohérence vis-à-vis de leurs besoins. On évite ainsi de perdre du temps lors de l'accès à la structure pour des vérifications finalement sans intérêt.

- accepter plusieurs formats de fichiers externe impose une définition quasi-canonique des objets. Cette définition formelle est largement validée si cette contrainte peut être respectée pour un nombre important de format de fichiers. La capacité de pouvoir accepter de nouveau formats de fichier *a posteriori* — **modgen** pour la vue physique, **spice** pour la vue logique — le démontre.

  Cela a l'avantage d'assurer un comportement identique des outils quel que soit le format d'entrée choisi, car la référence est le modèle mémoire dont la sémantique est invariante.

  Ainsi tous les outils peuvent proposer de manière automatique de nombreux formats de fichiers, ce qui permet d'épargner en pratique les temps de conversions et l'espace disque nécessaire au stockage du même objet sous de multiples formats.

- utiliser le langage **C** résoud l'important problème de la portabilité sur de multiples architectures matérielles. Ce langage est très performant sur une vaste gamme de machines, standardisé par un comité indépendant ce qui en garantit la longévité et partie intégrante des systèmes UNIX actuels. Ces trois atouts garantissent, du point de vue du langage, la pérennité de la structure de données.

En conséquence, nous avons prouvé que ces structures de données fournissent une plate-forme de développement structurée améliorant sensiblement la productivité des programmeurs et garantissant la portabilité et l'interopérabilité avec les systèmes de **CAO** majeurs.

La définition d'objets fortement typés et structurés, conforme pour la vue physique à l'approche symbolique sur grille fixe décrite au chapitre 1, rend cette structure bien adaptée au développement des générateurs procéduraux indépendants de la technologie. Les fonctions de bas niveau forment une base bien adaptée à l'écriture de macro-fonctions dédiées à la description de blocs par programme.

Chapitre 3

Un langage pour l'écriture de générateurs paramétrables

3.1 Introduction

L'objectif général de la **CAO** pour *VLSI* est de fournir des circuits performants à partir de spécifications de haut niveau[JM84]. Ce niveau d'abstraction est recherché par des outils connus sous le nom de *compilateurs de silicium*, terme introduit en 1979 par Johannsen et Mead, chercheurs travaillant sur Bristel Blocks[Joh79], la première ébauche d'un tel outil. Les compilateurs de silicium intègrent essentiellement deux phases, une phase de construction architecturale, dite de synthèse de haut niveau, qui à partir des spécifications génère essentiellement des interconnexions entre des automates d'états finis pour les parties contrôle, des chemins de données et des mémoires, et une phase de génération qui à partir des comportements de chacun de ces sous-ensembles produit la vue physique[Wer83].

Les automates sont par nature très irréguliers, et sont synthétisés par des outils dits de synthèse logique qui utilisent des bibliothèques de cellules précaractérisées. Au contraire, les composants d'un chemin de données et les mémoires possèdent une régularité intrinsèque, qu'on peut exploiter pour obtenir des circuits denses et rapides. Ceci est possible quand les bibliothèques contiennent des générateurs de blocs réguliers paramétrables. De tels générateurs nécessitent d'une part un langage de description procédurale, *silicon assemblers*[Sch87] ou *tiling languages*[CM88] selon les fonctionnalités, et d'autre part des bibliothèques de cellules de base spécifiques à chaque générateur.

L'écriture de programmes paramétrables pour la génération de blocs ne se justifie que si les générateurs sont portables — indépendant d'un processus de fabrication ou d'une chaîne de **CAO** particulière — et performants — en terme de surface et de temps de propagation —, car c'est un développement coûteux. Écrire un programme paramétrable de génération de circuits est une tâche multidisciplinaire. Un tel code demande des compétences dans les domaines :

- de la définition et de la mise en œuvre d'algorithmes pour les opérateurs arithmétiques.
- de la programmation et des techniques de développement logiciel, y compris la

vérification du code.

- de la circuiterie et une connaissance des problèmes électriques qui y sont liés.
- des aspects topologiques dus au découpage en cellules de bases et à leur assemblage par aboutement ou par routage.
- des techniques de vérification et de caractérisation des objets *VLSI* générés.

Deux types de générateurs sont utiles :

- les générateurs de blocs topologiquement très réguliers, comme les *ROM*s, les *RAM*s, les bancs de registres multi-accès, ou les opérateurs pour l'arithmétique entière. Dans ce cas, c'est un dessin obtenu par aboutement optimal de cellules de base qui est produit.
- les générateurs de blocs ne possédant pas une grande régularité topologique, mais basés sur des algorithmes garantissant de bonnes performances, typiquement les opérateurs pour l'arithmétique flottante. C'est alors un schéma de portes utilisant une bibliothèque de cellules précaractérisées qui est produit.

Après un état de l'art des langages de génération de modules, nous verrons en quoi le langage GENLIB permet une approche plus efficace de la génération procédurale.

3.2 Langages de description de circuits : état de l'art

3.2.1 Classement des langages

Les langages de génération se classent en deux grandes catégories :

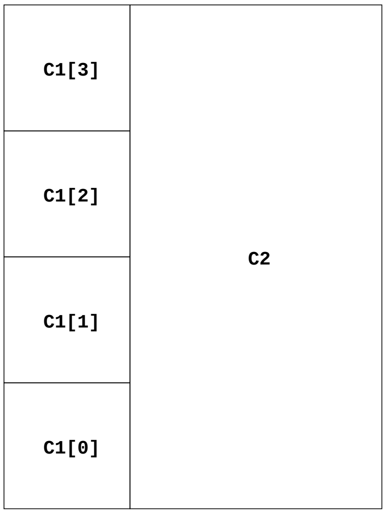
- les langages spécialisés, c'est à dire ceux pour lesquels un interpréteur ou un compilateur dédié est nécessaire. Ils ont une syntaxe spécifique, même s'ils utilisent des constructions ou des mots clefs proches de langages connus.
- les langages de programmation standards enrichis à l'aide de fonctions dédiées à la manipulation des objets *VLSI*. Les outils standards sont utilisés pour la chaîne de compilation.

Tous les langages actuels possèdent des fonctions permettant de décrire des interconnexions entre blocs, puis ensuite l'appel de routeurs intégrés pour résoudre des problèmes d'interconnexions non pris en compte lors de la définition des interfaces entre sous blocs.

3.2.2 Les langages *ad hoc*

Des chercheurs en intelligence artificielle du **MIT** ont proposés le langage **DPL**[BH80] basé sur une syntaxe à la **lisp** qui fait figure de précurseur dans ce domaine. D'autres langages issus de la recherche universitaire ont vu le jour ensuite. Le langage **ALI**[LRS$^+$82] dont la syntaxe est basée sur le **pascal**. Son but est de décrire à la fois les cellules de base et les

blocs hiérarchiques avec la même syntaxe. Cette idée sera d'ailleurs reprise dans presque tous les langages actuels. Les primitives de description sont assez archaïques, et leurs fonctionnalités assez limitées, surtout dans le maniement de la hiérarchie.
Le langage **ABCD**[RW82] est basé sur une syntaxe exotique, mais introduit la notion de placement relatif d'instances. La figure 3.1 illustre l'exécution de quelques instructions de ce langage relativement primitif. Ces langages n'ont plus aujourd'hui qu'un intérêt historique.




```
c1: instance gate1 ll = (0, 0)
    num = 4 dir = vertical
c2: instance gate2 ll = c1[0][lr]
```

Figure 3.1: Placement d'instances dans le langage **ABCD**, (d'après[WE85].)

Toutes les chaînes de **CAO** industrielles offrent un langage spécialisé pour l'écriture des générateurs.

COMPASS DESIGN AUTOMATION fournit pour cela le langage **VIP**[Com91], *VLSI Implementation Program*.
Les objets manipulés sont les objets du format **cif**[NS79]. Ce sont typiquement des polygones, des rectangles, des segments et des instances. Le langage est une interface procédurale dédiée au langage **cif**, et ne sait générer que des fichiers dans ce format pour une technologie donnée.
Les cellules peuvent être décrites procéduralement, ou saisies sous un éditeur de *layout* au micron indifféremment. La méthodologie encouragée consiste cependant à tout décrire par programme.
VIP permet une description de la hiérarchie grâce à des opérateurs de juxtaposition. Ces opérateurs agissent soit vis-à-vis de connecteurs des instances, soit par rapport à des points prédéfinis par l'utilisateur, soit encore par rapport à une enveloppe par défaut. Cette enveloppe utile pour les aboutements est définie par les connecteurs de chacune des faces.
Le langage est en fait un ensemble de fonctions basées sur le langage **Mainsail**[WDJ79], un langage interprété initié à Stanford proche du **pascal**. Le langage est syntaxiquement assez exotique, et la forte dépendance technologique induite par le choix de **cif** comme base de représentation des objets en limite l'intérêt.

La société MENTOR GRAPHICS CORPORATION propose le système **GDT**, *Generator Development Tools*, basé sur le langage **L**[Men90]. Les entités manipulées dans cet environnement pour la description des générateurs sont :

- transistors.
- contacts.
- connecteurs.
- segments.
- instances de cellules.
- matrices d'instance de cellules.
- rectangles et polygones.
- et les attributs attachés à chacun de ces objets.

L'assemblage des hiérarchies ainsi que la description du *layout* des cellules de base se font par programme. Les générateurs de cellules de base utilisent des règles topologiques — règles de distances entre niveaux, largeurs minimum, *etc* — pour décrire le placement relatif des objets de bas niveau. Ainsi, au lieu d'exprimer une distance entre deux objets par une valeur numérique, c'est l'évaluation d'une fonction dépendant des règles de dessin qui est utilisée. Cette approche est censée garantir l'indépendance technologique, car elle introduit les règles de dessin sous forme de variables. Néanmoins, on peut sans peine imaginer la complexité de la description textuelle d'une cellule — soixante lignes de **L** pour un inverseur —, avec des calculs dépendants des règles de dessin. Il y a plusieurs formules possibles pour décrire procéduralement une même cellule pour une technologie donnée, et le choix des formules les plus pertinentes, pour une meilleur densité par exemple, n'est pas simple.

En langage **L**, le placement des instances se fait en majorité grâce au verbe `AT`. Sa forme la plus générale est : *objet* `AT` *objet* `+/- (` *expr* `,` *expr* `);` L'idée est d'utiliser un des connecteurs de chacune des instances à placer pour définir l'aboutement. Une relation d'équipotentialité est alors établie entre tous les connecteurs qui sont en contact sur les faces concernées des instances, qui est conservée si l'une des instances devait être déplacée dans la suite du générateur, pour ouvrir un canal de routage par exemple. On a du mal à imaginer que la description procédurale de deux cellules adjacente va être telle que pour deux technologies différentes, tous les connecteurs considérés des deux cellules vont coïncider. Pour cela, il faudrait garantir que ce sont les mêmes règles de dessin affectées des mêmes coefficients qui sont utilisées pour les deux faces aboutables de deux cellules. La contrainte sur la conception est très forte, car une cellule s'aboute souvent avec plusieurs autres.

L propose également l'utilisation d'un compacteur en corrélation avec la description procédurale. Dans ce cas les cellules de base peuvent être décrites au micron pour une technologie fixée, et l'appel du compacteur permettra théoriquement de s'adapter sur d'autres technologies. Mais pour cela, en plus du dessin des primitives, il faut exprimer pour chacun des connecteurs de chacune des cellules les contraintes d'alignement avec les connecteurs des instances qui y sont logiquement reliés.

L'utilisation de ces techniques simplifie les modifications à apporter sur le code d'un générateur pour l'adapter à une nouvelle technologie [Men90]. Elles n'apportent pas une réelle indépendance vis-à-vis des procédés de fabrication. Il n'y a pas de solution générale portable aux problèmes d'aboutement avec **L**.

Le langage **L** est inspiré du langage **C**. Le langage en lui même est une copie presque conforme du langage **C** pour ce qui concerne les structures de contrôle, c'est à dire les boucles, les tests, . . ., et de sa bibliothèque standard, pour les fonctions de gestion des noms, d'affichage, de manipulation de fichiers. Les types utilisables sont ceux de base du **C**, renommés pour l'occasion. Un certain nombre de constructions spéciales ont été ajoutées pour le placement, avec une syntaxe tres différente, ce qui nuit à l'homogénéité de l'ensemble. Un exemple de programme **L** (d'après [Bur88]) :

```
CELL addergen(bits, pitch)
{
   NUM i = 0;
   CALL adderbit CELL bit(pitch);
   WHILE (++i <= bits) {
      INST bit bit[i];
      IF (i > 1) {
         bit[i].vdd AT bit[i - 1].vdd;
      }
   }
}
```

`CELL` définit la fonction `addergen` qui crée des masques. `CALL` va appeler une fonction `adderbit`, qui créera un modèle de nom `bit` pour le paramètre considéré. `INST` déclare une instance de nom `bit[i]`, de modèle `bit`, et la place par défaut à la coordonnée (0, 0). Le placement effectif se fait ensuite à l'aide de `AT`, qui vient repositionner l'instance `bit[i]`.

Il est intéressant de constater que la structure de données sous-jacente au langage, la **L Data Base**, est elle développée en **C** et est directement accessible par l'utilisateur à travers des fonctions **C** documentées. **GDT** est issu du système **Genesil**, de SILICON COMPILER SYSTEMS, qui est écrit en **C**. Pourquoi les concepteurs du système n'ont-ils pas poussé leur logique jusqu'au bout, et fait de **L** un surensemble du **C**?

Le langage, dans l'environnement **GDT**, est utilisé dans l'industrie pour la conception digitale, chez PHILIPS ou chez THOMSON. Notons que ces deux sociétés possèdent leurs propres lignes de fabrication.

Le langage **Modgen** est un autre langage basé sur le **C**. Il a été développé par PHILIPS. **Modgen** a été choisi afin d'échanger le code de générateurs digitaux au sein des partenaires du projet européen **IDPS**[IDP90]. Très proche du langage **L**, il en hérite malheureusement les défauts. Les cellules de base sont décrites textuellement en **Modgen**, en fonction de variables contenant les valeurs des règles de dessin. L'usage conjugué de ces variables et d'un compacteur tente de résoudre les problèmes de portabilité technologique. Il faut pour cela décrire la connectivité symbolique , et les contraintes d'aboutabilité entre instances. Le concepteur d'un générateur doit donc décrire à la fois la vue physique et une pseudo vue logique pour fournir un *layout*.

Syntaxiquement, le langage est une redéfinition de l'ensemble des structures de contrôle, des types et fonctions standards du **C**. Il possède de plus des limitations gênantes, comme l'impossibilité de pouvoir déclarer des tableaux de noms, où de faire des appels récursifs, ce qui peut être pénible pour certains générateurs[LGT93] dont l'algorithme de construction est naturellement récursif.

La chaîne CADENCE OPUS propose un langage unifié, **Skill**, pour permettre l'accès à ses structures de données internes[Cad91b]. Ce langage est bien plus qu'un langage de génération procédurale dans le sens ou il permet la construction de nouvelles applications, et également la modification du comportement de celles existantes, pour l'ensemble des fonctionnalités contenues dans CADENCE. C'est un langage de programmation à part entière, à la différence de **L** ou **Modgen** qui sont spécifiquement orientés génération de circuits. Afin de permettre une utilisation plus simple pour la génération procédurale, un ensemble de fonctions de plus haut niveau, celle du **ModuleMaker**[Cad91a] est défini. La portabilité technologique est envisageable, par l'utilisation de cellules saisies en *stick* et d'un compacteur. Les cellules sont dessinées sous un éditeur symbolique, et sauvegardée sous formes de procédures **Skill** que l'on peut venir modifier manuellement. Ces procédures sont alors intégrées dans le générateur, qui les assemble à l'aide de fonctions **Skill**. Il faut alors définir les contraintes d'environnement de chacune des cellules, afin

d'assurer le respect des aboutements après compaction. Il n'y a pas moins de trois options de compaction, et cinq types de contraintes définissables, ce qui témoigne de la complexité de la méthode.
Si la portabilité sur divers procédés de fabrication n'est pas envisagée, alors des fonctions permettant le placement relatif d'instances par rapport à une boîte d'aboutement sont accessibles. Elles sont cependant très orientées vers la description de matrices.

Le petit exemple de placement serait :

```
mmDefinePCell("addergen"
 ((bits 4) (pitch 6))
  i = 0
  bit = dbOpen("bit layout")
  while((i < bits)
     sprintf(iname "bit[%d]" i)
     if((i == 0)
        mmPlaceCell(bit list(0 0)
                    iname "R0")
     else
        sprintf(sname "bit[%d]" (i - 1))
        mmPlaceCell( bit
          mmAlignPoint(
            mmInstPinLoc("vdd" sname)
            mmPinLoc("vdd"))
          iname "R0")
     )
     i = i + 1
  )
)
```

Une cellule paramétrable est définie par `mmDefinePCell`, et ses paramètres par défaut sont définis. Un modèle est récupéré dans la base de données par `dbOpen`. Et finalement le placement se fait grâce à `mmPlaceCell`, fonction dont on peut spécifier le mode de placement par de multiples manières, soit en absolu, soit relativement a un point d'un objet précédemment placé, tel un connecteur d'instance.

La syntaxe du langage est inspirée *à la fois* de **C** et **lisp**. **Skill** est un langage généraliste, axé sur la manipulation de listes, c'est son coté **lisp**, mais qui permet de faire des boucles et des tests simplement, grâce au **C**. Il se base sur une bibliothèque complexe, de plus d'un millier de fonctions, et s'adresse, dans sa version la plus générale, non pas aux concepteurs de circuits mais aux développeurs d'outils de **CAO**. L'environnement OPUS comprend deux outils intéressants pour le développement de générateurs :

- un outil de saisie graphique de l'algorithme d'un générateur, *Design By Example*. Il permet de faire du placement sous éditeur graphique en spécifiant des opérateurs de condition, de répétition, de contraintes. Néanmoins, il n'est utile que pour la première phase de conception de générateurs très réguliers, et produit un squelette de code qu'il faut impérativement modifier.
- un outil de mise au point interactif[WL86] des programmes **Skill**. Il permet l'exécution en pas à pas des procédures l'utilisateur.

Skill est notamment utilisé chez SGS-THOMSON, une société maîtrisant sa propre technologie.

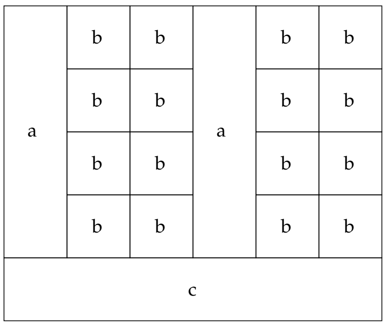
3.2.3 Les langages enrichis

Une autre direction, assez naturelle, est de choisir un langage généraliste, dont la syntaxe est documentée et figée, et de l'enrichir de fonctions spécifiques à la manipulation des objets utilisés pour la conception.

Compiler Development System, l'environnement de conception de blocs paramétrables liés au compilateur de silicium Concorde[Cas93], utilise le langage **C**.
La construction de blocs avec **SLIC** est dirigée par les règles de dessins. La portabilité technologique est envisagée par une description textuelle des cellules de base utilisant des informations technologiques paramétrables. Toutes les dimensions dans un fichier **SLIC** sont exprimées en fonction de ces variables. La méthode utilise des marqueurs, *landmarks*, qui forment une grille relative, (voir page 4), pour le placement de primitives. Chaque transistor, contact, fil, est placé en fonction de ces marqueurs dans les cellules de base. Cette grille est ensuite utilisée pour la totalité du générateur, tous niveaux hiérarchiques confondus. Cette grille molle varie en fonction de la valeur des règles, et ainsi l'ensemble de la hiérarchie est théoriquement adaptable sur plusieurs technologies.
Le placement hiérarchique se fait relativement aux connecteurs ou aux enveloppes — *bounding boxes* — des instances. La position des connecteurs et l'encombrement d'une cellule sont variables avec les règles de dessin. Il faut donc décrire *n* cellules s'aboutant en prenant soin d'utiliser des formules identiques pour les *landmarks* qui définissent les connecteurs en coïncidence. La programmation en résultant est d'une grande complexité. De plus, les mêmes restrictions sur la validité à travers les technologies d'une description procédurale soulevées pour le langage **L** se retrouvent ici.

Les fonctions de **SLIC** supposent connues en **C** les notions de pointeurs, de types de données et d'accès aux membres de structures. Ces connaissances sont d'une utilité discutable pour le problème visé.

Le langage **Lubrick**[Sch85], développé à l'**IMAG**, est un surensemble du **pascal**. La portabilité technologique des générateurs n'est pas un des objectifs primordiaux de **Lubrick** qui avait pour but l'assemblage de chemins de données optimisés.
Les cellules de base sont saisies sous un éditeur graphique au micron, sous forme de rectangles et de trapèzes. Elles sont ensuite assemblées grâce au langage.




```
var i,j,k,l:pointeur;
begin
  i:=openfig('fp');
    j:=openfig('fi');
      j:=right(j,getfig('a'),1,0);
        k:=repy(getfig('b'),4);
      j:=right(j,k);
    j:=closefig(j);
    l:=repx(j,2);
    i:=up(i,getfig('c'),1,0);
    i:=up(i,l,1,0);
    i:=closefig(i);
end;
```

Figure 3.2: Placement d'instances en **Lubrick**, (d'après[Sch85]).

Il permet le placement relatif d'une instances par rapport à l'enveloppe du modèle en cours de construction. (Chacun des modèles possède une enveloppe polygonale définie avec des rectangles d'un niveau de masque virtuel.) Les règles de croissance de l'enveloppe de la figure en cours de construction doivent être parfaitement connues lors de l'écriture du générateur, et l'on ne peut placer une instance qu'à droite ou au dessus de l'enveloppe de la figure. Cette approche très particulière nécessite la création de niveaux hiérarchiques uniquement pour pouvoir faire le placement, ce qui ne correspond pas forcément au découpage désiré par le concepteur. Des opérateurs de répétition permettant de dupliquer *n* fois une instance sont disponibles. Un exemple de placement à l'aide de ce langage est donné figure 3.2.

Le langage ne permet pas d'appliquer un opérateur géométrique à une instance, mais il permet de créer un modèle qui a subit une transformation à partir d'un modèle original. Cette approche simplifie la récupération des connecteurs des niveaux hiérarchiques inférieurs pour l'outil, mais n'est pas très naturelle pour le concepteur car elle introduit des modèles non significatifs.
Le langage **Lubrick** suppose connues les notions de pointeurs, car chaque fonction renvoie un pointeur qui sera passé à une autre fonction. L'utilisation de **pascal** comme langage sous-jacent est un choix raisonnable. C'est un langage simple et pédagogique et c'était à l'époque le seul langage fortement typé.

Le langage **Styx**[CFJ93], développé au sein de SGS-THOMSON, se fonde sur **le lisp**. Du point de vue de ses choix de principes pour la description des masques et des hiérarchies, le langage est très proche de **L**. La portabilité des cellules de base est envisagée par l'utilisation de formules incluant les règles de dessin sous forme de variables. Des mini-générateurs créent un ou des transistors avec diverses configurations d'aboutement, qui sont utilisés pour la construction des cellules, ce qui est censé assurer l'indépendance technologique. Les cellules de base peuvent être saisies sous un éditeur graphique et sauvées sous forme de programme **Styx**. Elles sont ensuite éventuellement modifiées pour exprimer des distances entre objets sous forme de formules paramétrées, ce qui simplifie la tâche des concepteurs. Néanmoins le choix des bons paramètres à utiliser reste difficile à définir. Des interconnexions peuvent être décrites sous forme de liste de connecteurs d'instances, qui peuvent alors être routées. Mais il n'y a pas dans ce langage de primitives pour la description de *net-list*s de cellules, pour des opérateurs en cellules précaractérisées. **Styx** a été adapté au dialecte **lisp** de la chaîne CADENCE, ce qui permet le développement et l'exécution des générateurs dans un environnement intégré. Ceci permet aux utilisateurs, dans le cadre industriel de SGS-THOMSON, de travailler dans un monde unifié qu'ils connaissent bien. Le langage hérite des facilités de mise au point du **lisp**, qui est un langage interprété : on peut visualiser la valeur des variables et les appels de fonctions en cours de déroulement d'un programme. Néanmoins, comme le concepteur n'utilise normalement que peu de variables, et que celles utiles au placement sont gérées implicitement par le système, ça ne remplace pas un outil de mise au point graphique interactif.

L'inconvénient majeur du langage vient de la complexité de description des cellules servant dans l'assemblage, car elles dépendent explicitement des variables contenant les règles de dessin. Les restrictions sur la portabilité sont les mêmes que celles du langage **L** évoqué précédemment. Les contraintes drastiques de la méthode sur les cellules de base sont illustrées par l'exemple suivant : un générateur de **PLA** contenant 10 cellules de base est décrit en 1800 lignes de **Styx**. 700 lignes font le placement hiérarchique, et les 1100 lignes restantes décrivent les cellules de base.

Le **fortran** est également utilisé pour la génération de blocs. Un exemple est **MGL**, *Module Generation Language*, un langage défini et utilisé chez PLESSEY[Cla90]. Une portabilité technologique n'est pas envisagée pour les objets manipulés dans ce langage. PLESSEY possédant ses propres lignes de fabrications, une approche homothétique par rapport à une technologie de référence est suffisante.
Les cellules de base sont saisies sous un éditeur graphique au micron. Elles sont ensuite

assemblées grâce au langage. Il n'y a pas de problèmes d'aboutement, puisque les cellules de base sont dessinées pour pouvoir s'abouter sur la technologie choisie.
Il est possible de décrire des générateurs par aboutement et des générateurs d'interconnexions grâce à **MGL**, ces deux approches étant complètement indépendantes.
La construction d'une figure se fait par placement absolu d'instances, mais grâce à des coordonnées qui peuvent être récupérées sur des instances qui ont déjà été placées. Ceci permet le placement relatif d'instances, en spécifiant les deux instances à chaque appel.
MGL fournit un ensemble suffisant de fonctions pour la génération procédurale, et a été utilisé pour décrire un nombre important de générateurs dans l'environnement de conception interne à PLESSEY.

3.2.4 Critique des approches existantes

Dans toutes ces approches, le problème de la portabilité sur des procédés de fabrication variés est soit non traité, soit complexifie d'un ordre de grandeur la description d'un générateur. La portabilité technologique est envisagée dans les approches existantes grâce à :

- un compacteur. Dans ce cas, il faut décrire, en plus du placement et en même temps, des contraintes d'alignement entre connecteurs d'instances et parfois des *netlist*s. Si la complexité de l'algorithme de placement est liée au nombre d'instances, l'algorithme servant à décrire les contraintes ou les interconnexions sera plutôt dépendant du nombre de connecteurs d'instances. Le nombre de connecteurs d'instances est au moins dix fois supérieur au nombre d'instances, ce qui indique clairement que l'écriture d'un générateur dans ce contexte est beaucoup plus compliqué.

  Devoir décrire en même temps que le placement les interconnexions entres les connecteurs d'instances est inutile et parasite pour les blocs construits par aboutement. Ceci fait reposer sur le concepteur d'un générateur le problème de la portabilité parce qu'il est mal résolu par l'environnement de conception. Plus un programme est complexe, plus il est sujet à erreurs. Les difficultés liées à la conception d'un générateur doivent résider dans le choix d'une architecture ou de la circuiterie plutôt que sur son implémentation en tant que programme.

- une description textuelle basée sur des variables dépendant de la technologie dite *dynamic virtual grid compaction*[CS88]. La description des cellules est certainement pénible, et peu naturelle au concepteur de circuit pour qui une cellule est avant tout une représentation graphique. L'aspect visuel permet d'intégrer les contraintes d'aboutement entre cellules relativement simplement. Réussir à transformer cette description graphique en un programme paramétrable est une tâche délicate. Il faut, pour programmer les masques d'une cellule, choisir les bonnes variables de distances, qui assureront la portabilité effective. En effet, si pour une technologie donnée c'est telle règle de dessin qui impose la distance dans telle direction, il n'en sera pas forcément de même pour une autre technologie. C'est un problème de dépendance contextuelle que tentent de résoudre les compacteurs pour une technologie donnée, et il est prétentieux de croire que l'on puisse le résoudre procéduralement pour un ensemble de technologies[McG93].

  L'aspect homogène de cette approche, où la totalité d'un générateur est décrite sous la même forme, celle d'un programme, est séduisant mais peu naturel. Elle ne correspond ni à un besoin, car il est beaucoup plus simple et intuitif de saisir les cellules de base graphiquement, ni à une nécessité, car elle n'assure pas pour autant la portabilité effective. Elle n'est pas adaptée à la représentation de la structure des circuits, où le dessin des cellules est une activité très différente de l'assemblage des hiérarchies. Cette approche est plus un fantasme de programmeur qu'une méthode réellement adaptée à un concepteur en micro-électronique.

Une critique générale sur les langages *ad hoc* concerne leur disponibilité et leurs possibilités d'ouverture :

- ils utilisent une syntaxe spécifique, donc exigent un apprentissage important.
- ils ne sont exécutables que dans le contexte de leur chaîne de **CAO**, ce qui rend les bibliothèques de générateurs dépendantes du fournisseur de la chaîne de **CAO**.
- étant tous basés sur un interpréteur, leur rapidité d'exécution est en général très inférieure à celle du même algorithme compilé.

Le point commun de presque tous ces langages *ad hoc* est qu'ils utilisent une syntaxe et des structures de contrôles proches de celle de langages standards, comme le **pascal**, le **lisp** ou le **C**. La conclusion qui s'impose est que l'écriture d'un générateur ne nécessite pas l'introduction de structures de contrôles particulières pour coder les algorithmes.

Certain de ces langages nécessitent des notions de programmation complexes, pointeurs, structures, *etc*. Ces connaissances ne sont pas indispensables à la description du placement d'instances, et sont des éléments de confusion. De même, il est important de pouvoir disposer d'outils d'aide au développement logiciel des générateurs, et seul **Skill** propose cette fonctionnalité parmi les langages étudiés.

La description procédurale de la vue structurelle n'existe pas en tant que telle dans les langages de génération usuels, hormis **MGL**, car l'idée de faire des générateurs paramétrables utilisant comme cible une bibliothèque de cellules précaractérisées est récente[Fra92]. Cette approche commence à être utilisée industriellement : option portable des générateurs de COMPASS. L'utilisation de cellules précaractérisées est essentiellement réservée à la logique sauvage, généralement synthétisée à l'aide d'outils à partir du comportement. Malheureusement, de tels outils ont leurs limites et ne conviennent pas pour des structures algorithmiques ou de mémorisation comme celles que nous visons.

Tout cela conduit à la définition d'un langage original, GENLIB.

3.3 Principes et objectifs du langage

GENLIB doit satisfaire aux principes suivants :

1. **portabilité technologique.** Les blocs générés doivent être portables sur un vaste ensemble de technologies. Les contraintes de *pitch matching* dans les deux directions x et y font du *layout* sur grille fixe sans compaction l'approche la mieux adaptée pour la description des cellules dédiées aux générateurs par aboutements.

2. **séparation cellules de base / placement.** La méthode de dessin des cellules de base et la méthode d'assemblage hiérarchique sont différentes. Les cellules de base sont dessinées sous un éditeur graphique de *layout* symbolique. L'important niveau d'abstraction fourni par la méthode assure une bonne productivité. La complexité du dessin d'une cellule de base est beaucoup plus facile à appréhender visuellement, et c'est l'approche naturelle d'un concepteur de circuit. Les cellules sont ensuite assemblées en hiérarchies grâce au langage. L'utilisation de boîtes d'aboutement rectangulaires assure la simplicité du modèle d'assemblage pour le placement des instances.
3. **séparation de la vue logique et de la vue physique.** La description de la vue physique et la description de la vue logique ne sont pas liées. Il est inutile de compliquer le développement d'un générateur en décrivant deux algorithmes séparés mélangés dans le même code. Si ces deux vues sont utiles, elles seront générées indépendamment.
4. **utilisation de la hiérarchie.** Il faut être capable de décrire la hiérarchie. C'est un concept puissant qui permet de simplifier la gestion de la complexité.
5. **utilisation d'un langage standard.** Le langage est un ensemble de fonctions basé sur un langage généraliste. Les langages standards assurent la portabilité logicielle.
6. **indépendance vis-à-vis des systèmes de CAO commerciaux.** Il faut cependant être capable de s'interfacer avec eux.
7. **simplicité du langage.** Le langage de programmation doit être simple à aborder pour les non-programmeurs.

La conception d'un générateur est d'abord un travail de micro-électronicien, de concepteur de circuits intégrés, et non un travail d'informaticien, de programmeur. Les principes à la base de la définition du langage GENLIB sont le reflet de cette conviction.

On se pose comme objectifs de pouvoir décrire aisément et efficacement avec GENLIB deux types de générateurs :

- des générateurs physiques paramétrables utilisant une technique d'aboutement. La méthodologie adoptée se base sur une bibliothèque de cellules dédiée à chaque générateur, et sur un programme d'assemblage décrivant de façon hiérarchique un placement topologique utilisant cette bibliothèque. Les cellules de base, *leaf cells*, ne sont pas décrites en GENLIB, mais dessinées manuellement sous un éditeur graphique en *layout* symbolique. Le programme d'assemblage, le *tiler*, est écrit en GENLIB.
- des générateurs logiques paramétrables. On se base sur une bibliothèque générique de cellules précaractérisées qui est partagée par tous les générateurs. Le placement des cellules est fait soit en GENLIB, de manière un peu similaire à l'approche cité ci-dessus, soit par un outil de placement automatique externe standard laissé au choix de l'utilisateur. Le routage est toujours fait par un outil externe.

Le langage GENLIB repose sur les choix de principes adoptés pour la structure de données MBK décrits au paragraphe 2.2.

3.4 Choix du langage C

Pour le concepteur de générateurs, les bénéfices du **C** sont :

1. utilisation d'un langage standard. Le **C** est enseigné dans toutes les universités, et les ouvrages qui en expliquent la syntaxe et l'utilisation sont légions.
2. accès à des outils de développement puissants qui dépassent le cadre de la conception *VLSI*. Le langage **C** est utilisé dans de nombreux domaines, et les outils d'aide au développement et à la mise au point sont nombreux et efficaces.
3. c'est un langage qui peut être utilisé à plusieurs niveaux de complexité, et GENLIB propose son utilisation au niveau le plus élémentaire. En contrepartie, l'utilisateur averti pourra toujours profiter de l'ensemble des fonctionnalités du langage.

Du point de vue du développeur du langage, choisir **C** a de multiples avantages également :

1. il n'est pas nécessaire de reécrire un compilateur complet. Les compilateurs **C** disponibles aujourd'hui sont très fiables. Vu la complexité d'un tel développement, autant profiter des outils existants.
2. le langage est documenté et maintenu par des comités d'experts[ANS89]. Il n'y a pas de risques de changements majeurs qui pourraient le rendre caduc.
3. l'ajout de fonctionnalités, sous forme de fonctions, est d'une grande facilité.
4. le **C** est le plus portable, en regard des architectures matérielle possibles, des langages actuels. Ceci permet de toucher une vaste population de concepteurs.

C'est essentiellement l'aspect portabilité qui fait préférer le langage **C** à des langages comme **pascal** ou **lisp**.

3.5 Présentation générale de GENLIB

Afin de faciliter la prise en main par les concepteurs inexpérimentés en programmation, un sous ensemble du langage **C** est proposé. Ce sous ensemble est très simplificateur, tout en restant suffisant pour l'écriture des générateurs. Il n'est pas obligatoire de se limiter à ce sous ensemble : un programmeur expérimenté peut s'il le désire utiliser l'ensemble des constructions syntaxiques du langage **C**.

La principale restriction concerne le type des données utilisées, qui sont soit des valeurs numériques entières — pour la manipulation des nombres, indices de boucles, coordonnées, *etc* —, soit des chaînes de caractères explicites — pour nommer les objets —. La notion de pointeur, largement présente dans les applications **C** classiques, a été bannie en GENLIB. Toute référence à un objet se fait de façon symbolique par son nom, et non par un pointeur sur l'objet. La complexité algorithmique des générateurs étant faible, le surcoût en temps d'exécution est négligeable pour une facilité de programmation décuplée pour l'utilisateur.

Le langage GENLIB permet d'encapsuler les fonctions d'accès bas niveau aux structures de données MBK dans le but de fournir une interface procédurale. Les motivations qui ont prévalue lors de la définition de ces fonctions sont les suivantes :

- faire simple. Pour cela, les arguments des fonctions de GenLib n'ont que deux types d'arguments : des entiers et des chaînes de caractères.

  L'objectif est de masquer la notion de pointeur ainsi que la manipulation explicite des structures **C**.

- se donner la possibilité de modifier la structure de l'implémentation sans modifier la moindre ligne de code dans les générateurs existants. On peut par exemple imaginer des modifications dans l'organisation de la structure de données, ou même un portage de GenLib sur le langage **C++** si cela s'avère désirable.

- proposer des fonctions de haut niveau bien adaptées au problème. Les fonctions de recopie d'objets à travers la hiérarchie, ou de routage canal inter-blocs sont nécessaires dans le contexte de la génération procédurale, mais n'ont, sous cette forme en tout cas, pas d'intérêt autrement.

Nous allons présenter l'implémentation de la fonction GenLib `PLACE` afin de mettre en évidence les mécanismes d'encapsulation. Le but de cette fonction est de placer le coin bas/gauche d'une instance de nom `insname` d'une figure de nom `figname` aux coordonnées `x, y` après lui avoir fait subir l'opération géométrique indiquées par `sym`.

```
 1 void genPLACE(figname, insname, sym, x, y)
 2 char *figname;
 3 char *insname;
 4 int sym;
 5 long x, y;
 6 {
 7 phfig_list *ptfig;
 8 long deltax, deltay;
 9 num_list *ptnum;
10
11    if (WORK_PHFIG == NULL) {
12       (void)fflush(stdout);
13       (void)fputs("*** genlib error ***\n", stderr);
14       (void)fputs("PLACE impossible : missing DEF_PHFIG\n", stderr);
15       EXIT(1);
16    }
17    WORK_PHINS = addphins(WORK_PHFIG, figname, insname, sym, x, y);
18    ptfig = getphfig(figname, 'P');
19    if (sym == NOSYM || sym == SYM_X || sym == SYM_Y || sym == SYMXY) {
20       deltax = x + ptfig->XAB2 - ptfig->XAB1;
21       deltay = y + ptfig->YAB2 - ptfig->YAB1;
22    }
23    if (sym == ROT_P || sym == ROT_M || sym == SY_RP || sym == SY_RM) {
24       deltax = x + ptfig->YAB2 - ptfig->YAB1;
25       deltay = y + ptfig->XAB2 - ptfig->XAB1;
26    }
27    ptnum = addnum((num_list *)NULL, deltay);
28    ptnum = addnum(ptnum, deltax);
29    WORK_PHINS->USER = addptype(WORK_PHINS->USER, (long)PLACEABOX,
30                               (void *)ptnum);
31 }
```

Le prototype de la fonction, lignes 1 à 5, nous indique que la fonction prend 5 arguments. Les deux premiers sont des chaînes de caractères qui représentent respectivement le nom de modèle à instancier et le nom de l'instance. Le troisième représente la symétrie à appliquer à l'instance, et les deux derniers le couple de coordonnées indiquant le point d'appel de l'instance une fois la symétrie effectuée. Il faut noter que dans le contexte de la génération procédurale, il est agréable que le point d'appel d'une instance soit invariant par opération géométrique.

La ligne 11 vérifie que l'on place cette instance dans une figure préalablement déclarée. Le pointeur `WORK_PHFIG` est une variable globale qui pointe sur la figure en cours de construction. Sa valeur ne peut être modifiée que par un appel à la fonction définissant la figure courante `DEF_PHFIG()`.

C'est ligne 17 que se passe l'insertion de cette nouvelle instance dans la structure de données MBK. La fonction `addphins` retourne un pointeur sur la nouvelle instance. Le pointeur `WORK_PHINS` est une variable globale qui pointe sur l'instance de référence pour la prochaine fonction de placement relatif.

Les lignes 18 à 29 correspondent au calcul des coordonnées du coin en haut à droite de la boîte d'aboutement de la nouvelle instance et à l'introduction de cette information dans la structure de données — grâce au champs `USER` qui permet l'ajout d'informations dans les objets MBK—. Cette information est nécessaire pour les fonctions de placement relatif qui utilisent la boîte d'aboutement de l'instance de référence.

On voit sur cet exemple comment le langage GENLIB arrive à masquer les calculs fastidieux de gestion du placement relatif ainsi que les manipulations de pointeurs et d'accès à la structure de données.

Dans le descriptif qui suit, on expose dans les grandes lignes les primitives de GENLIB sans insister sur leur syntaxe. Une documentation exhaustive, qui présente chacune des fonctions sous l'angle de la syntaxe avec un exemple typique d'utilisation, est disponible dans [Pét92].

3.6 Générateurs utilisant une technique d'aboutement

Trois grandes classes de fonctions sont disponibles pour la description des blocs optimisés :

1. les fonctions de placement.
2. les fonctions de consultation.
3. les fonctions complexes.

La méthodologie suivie pour les générateurs de blocs optimisés se base sur une bibliothèque de cellules dédiées et un programme d'assemblage. La seule vue nécessaire des cellules de cette bibliothèque est la vue physique symbolique. Du point de vue du langage, les informations nécessaire à la manipulation de la cellule sont limitées et clairement identifiées, (voir figure 3.3).

Boite d'aboutement
c'est le rectangle utilisé par GENLIB lors du placement relatif des cellules.

Connecteurs
ils sont utilisés pour faire du routage inter-cellules, ou pour être remontés dans la hiérarchie. Il faut noter que l'index du connecteur est d'une importance primordiale, car c'est lui qui en conjonction avec le nom permet d'identifier de façon non ambiguë un connecteur dans le programme.

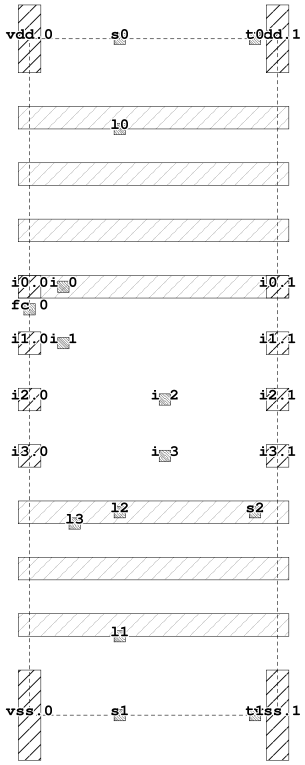


Figure 3.3: Interface d'une cellule attendu par GENLIB.

Références
: nécessaires pour le codage des cellules, comme connecteurs virtuels, ou utilisées au gré du concepteur.

Transparences
: elles permettent éventuellement la traversée d'un bloc par un fil de routage.

On notera que — à cause du concept de boîte d'aboutement —, GENLIB ne manipule que des instances rectangulaires et ne sait créer que des cellules rectangulaires. La figure 3.4 en est une illustration.

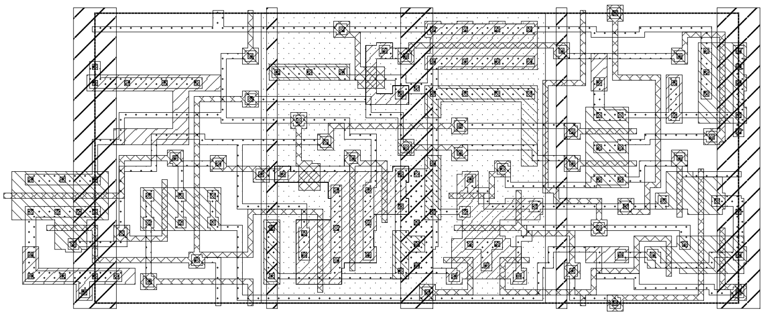


Figure 3.4: Une cellule typique d'un générateur optimisé, la boîte d'aboutement est matérialisée par le trait gras.

Dans ce contexte, les étapes identifiées de la construction d'un bloc sont :

1. définir la figure de travail, par l'intermédiaire de `DEF_PHFIG(NOM)`.
2. construire le placement des blocs, éventuellement de manière hiérarchique. Il est possible de créer plusieurs niveaux hiérarchiques dans un même programme GENLIB. On construit d'abord les niveaux les plus bas, que l'on instancie ensuite, sachant que le premier niveau est celui des cellules de base qui ne sont généralement pas décrites en GENLIB. Elles doivent être accessible sur disque afin d'en récupérer l'encombrement et l'interface.
3. spécifier la boîte d'aboutement de la figure, par un `DEF_AB(DX0, DY0, DX1, DY1)`. Les paramètres de cette fonction sont des écarts par rapport à la boîte d'aboutement standard, c'est à dire l'enveloppe rectangulaire minimale des boîtes d'aboutement de toutes les cellules instanciées.
4. définir l'interface du bloc généré — position des connecteurs, transparences, *etc* — en exportant les informations topologiques présentes dans les cellules instanciées.
5. sauver le résultat sur disque, grâce à `SAVE_PHFIG()`.

Les fonctions utiles pour les parties 2 et 4 sont décrites dans la suite.

3.6.1 Placement

La base de la construction de circuits paramétrables repose sur la capacité à placer des cellules les unes par rapport aux autres en assurant la continuité électrique aux frontières. Les fonctions de placement de GENLIB n'ont pas la vocation de garantir le respect de ces contraintes de connectique, car cela impliquerait la description explicite par le concepteur des interconnexions entre les cellules qu'il est en train de placer. Un des principes du langage étant la distinction entre la phase de dessin des cellules de base et la phase d'assemblage des cellules, il a été choisi de simplifier la tâche du concepteur en lui demandant de se focaliser sur l'algorithme de placement et de séparer la phase de construction de la phase de vérification. L'assemblage des instances se fait par l'appel aux fonctions :

`PLACE(MODÈLE, INSTANCE, SYMETRIE, X, Y)`
: qui permet le placement d'une instance à des coordonnées absolues spécifiées par l'utilisateur. Cette fonction ne devrait être utilisée que pour placer la première instance.

`PLACE_TOP(MODÈLE, INSTANCE, SYMÉTRIE)`
`PLACE_RIGHT(MODÈLE, INSTANCE, SYMÉTRIE)`
`PLACE_BOTTOM(MODÈLE, INSTANCE, SYMÉTRIE)`
`PLACE_LEFT(MODÈLE, INSTANCE, SYMÉTRIE)`
: sont destinées à placer une instance relativement à l'*instance de référence*. Par défaut, cette instance est la dernière instance placée.

`DEF_PHINS(INSTANCE)`
: permet de redéfinir l'instance de référence pour le placement relatif.

L'illustration du fonctionnement de ces primitives se trouve figure 3.5. L'instance centrale est l'instance sélectionnée comme instance de référence pour les appels successifs. Certains langages de génération possèdent une primitive permettant le placement par rapport aux

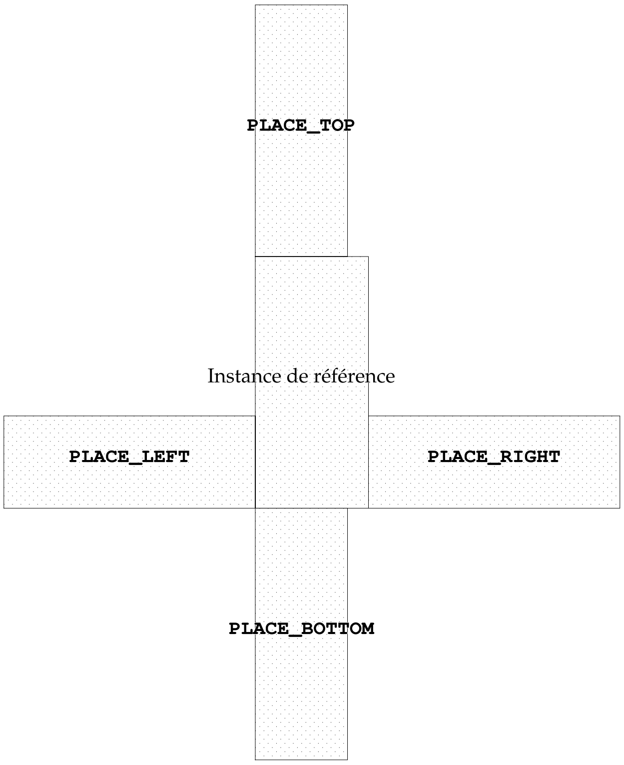


Figure 3.5: Les fonctions de placement relatif de GENLIB.

connecteurs d'instances. Cette approche n'a jamais été exploitée dans les développements en GENLIB, et l'étude des autres langages a prouvé que cette fonction avait pour but de palier l'absence de notion de boîte d'aboutement. En effet, ce sont dans 95% des cas les connecteurs d'alimentation situés dans les coins qui sont les paramètres de ce type de fonctions. De plus, garantir l'aboutabilité par un connecteur ne signifie pas la garantir pour le reste des connecteurs de la face, ce qui rend le placement inutilement complexe.

3.6.2 Récupération d'informations

Dans les générateurs de blocs optimisés, l'expérience montre que la moitié du programme est employé à la remontée hiérarchique des connecteurs, ou au placement de fils ou de contacts à partir d'informations contenues dans les cellules instanciées. Il est donc indispensable de fournir des primitives permettant de récupérer la position des objets d'un bloc fils, ou d'en recopier le contenu dans le bloc père efficacement. Sont disponibles des fonctions d'assez bas niveau pour tirer du placement des informations purement topologiques, sous la forme d'une coordonnée :

```
HEIGHT(FIGURE)
WIDTH(FIGURE)
```

permettent de connaître respectivement la hauteur et la largeur d'un modèle.

```
GET_INS_X(INSTANCE)
GET_REF_X(INSTANCE, RÉFÉRENCE)
GET_CON_X(INSTANCE, CONNECTEUR, INDEX)
```

qui permettent de récupérer l'emplacement d'une instance ou d'un objet d'une instance dans la figure. Ceci permet de faire du placement pseudo-relatif dans les rares cas où les fonctions de placement relatif ne sont pas adaptées.

Certaines primitives permettent la manipulation d'objet sans avoir explicitement recours aux positions dans le plan :

`COPY_UP_CON(INDEX, CONNECTEUR, INSTANCE, NOUVEAU_CONNECTEUR)`
Ceci copie un connecteur d'une instance dans le modèle en cours de construction. Si le connecteur de l'instance est situé sur la périphérie du bloc père, il est dupliqué tel quel. Dans le cas où il est situé à l'intérieur de la boîte d'aboutement du bloc père, alors un segment est tiré entre son emplacement et la boîte d'aboutement du bloc père. La figure 3.6 illustre ce comportement. Sur cet exemple, le connecteur `sg`

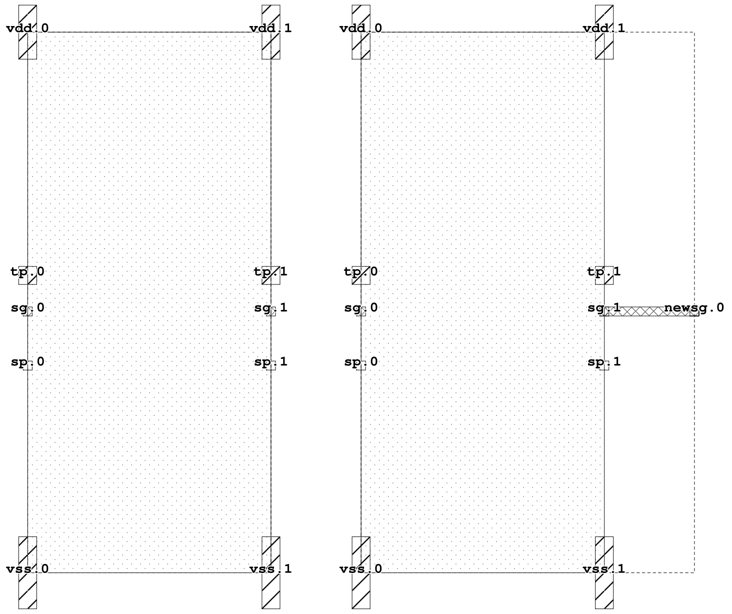


Figure 3.6: Effet de la fonction `COPY_UP_CON`.

d'index `1` de l'instance est copié sur la boîte d'aboutement du modèle sous le nom `newsg`.

`COPY_UP_REF(RÉFÉRENCE, INSTANCE, NOUVELLE_RÉFÉRENCE)`
`PLACE_VIA_REF(INSTANCE, RÉFÉRENCE, CONTACT)`
`PLACE_SEG_REF(INSTANCE, RÉFÉRENCE, COUCHE, LARGEUR, FACE)`
`PLACE_CON_REF(INSTANCE, RÉFÉRENCE, NOM, COUCHE, LARGEUR, FACE)`
parlent d'elles mêmes. En se basant sur les références présentes dans les instances, elles permettent le placement de divers objets dans le bloc père.

`WIRE(COUCHE, LARGEUR, INSTANCE1, CONNECTEUR1, INDEX1, ...)`
permet de réaliser un routage monocouche constitué de un à trois segments en se basant sur les connecteurs ou références d'instances. Cette fonction est très utile pour les générateurs complexes où du routage entre cellules doit être fait différemment suivant les configurations, et ne peut donc être codé à l'intérieur des cellules. On tient toujours à minimiser le nombre de cellules spécifiques à un générateur, pour des raisons de maintenance. Il est intéressant de noter que cela n'est possible qu'avec le dessin symbolique sur grille, ou avec une technologie fixe, mais que les systèmes à base de compacteurs ne peuvent généralement pas fournir ce genre de fonctions[Jan91].

3.6.3 Fonctions complexes

Dans le processus de construction d'un bloc, il est parfois utile d'avoir recours à des fonctions de haut niveau. Deux besoins ont été clairement identifiés.

`FLATTEN_PHFIG(INSTANCE, CONCATÉNATION)`
: recopie les objets d'une instance dans le niveau hiérarchique courant et efface l'instance. Le niveau hiérarchique `m0` est supprimé, et ses composantes sont recopiées

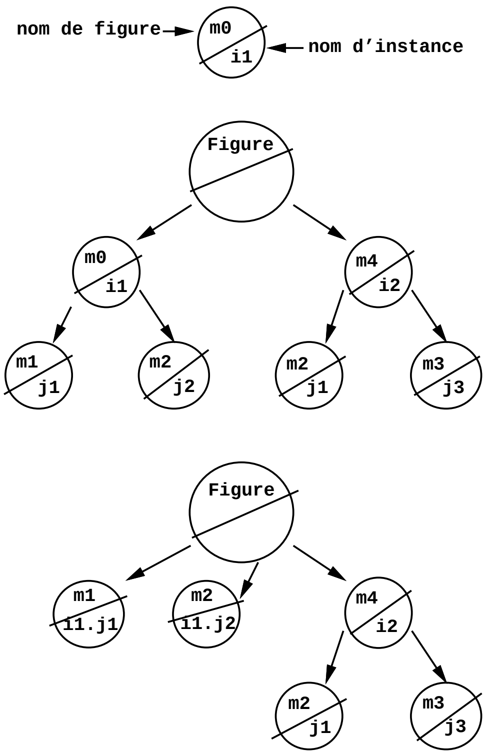


Figure 3.7: Actions liées à la mise à plat.

dans la figure. L'argument `CONCATÉNATION` indique si l'on veut que le nom des objets contenus dans le modèle soit concaténé avec le nom de l'ex-instance.

`ROUTE_CHANNEL(SERVITUDE, LARGEUR, ..., 0)`
: est un routeur canal deux couches : il utilise une *netlist* faisant référence aux connecteurs de deux blocs placés en vis-à-vis, décrite grâce aux primitives du paragraphe 3.8. Cette fonction n'est en pratique utilisée qu'au stade final de la conception, pour connecter électriquement deux blocs qui ne s'aboutent pas correctement. Un exemple typique est un banc de registres et son décodeur. Cette fonction prend comme arguments un nombre variables de servitudes — alimentations ou horloges — qui sont routées séparément du canal, car elles ont une largeur non minimale. Elle modifie la figure courante et fournit une figure rectangulaire qui est la figure courante routée.

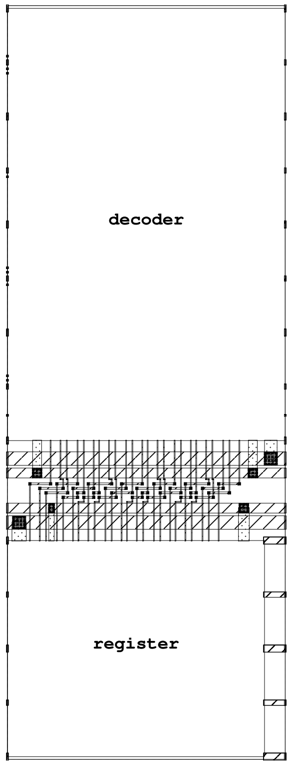


Figure 3.8: Routage d'un canal inter-blocs.

3.7 Fonctions pour la description procédurale de cellules

Bien que le langage ne vise pas en priorité la description procédurale de cellules de base, toutes les fonctions nécessaires sont cependant disponibles. La compilation de cellules a besoin de pouvoir créer tous les objets éditables sous un environnement graphique, en utilisant des coordonnées absolues, mais paramétrables. La fabrication de cellules automatiquement ne se conçoit que dans un contexte d'intégration dans une bibliothèque imposant certaines contraintes topologiques. Les contraintes d'aboutement sont réduites à leur plus simple expressions : continuité des alimentations et du caisson, hauteur fixe pour toutes les cellules, position des entrées sorties fixe.

Les primitives utiles dans ce contexte sont les fonctions de plus bas niveau de GENLIB :

`PHSEG(COUCHE, LARGEUR, SEGMENT, X0, Y0, X1, Y1)`
: crée un segment avec ses caractéristiques explicites.

`PHVIA(TYPE, X, Y)`
: pose un contact aux coordonnées spécifiées.

`PHREF(TYPE, RÉFÉRENCE, X, Y)`
: définit une référence.

`DEF_AB(X0, Y0, X1, Y1)`
: définit la boîte d'aboutement de la figure.

`PHCON(COUCHE, LARGEUR, CONNECTEUR, FACE, X, Y)`
: ajoute un connecteur dans la structure. Cette fonction ne peut être utilisée qu'après

une définition explicite de la boîte d'aboutement, puisque par convention les connecteurs doivent se trouver sur l'une des faces.

Ces fonctions ont été utilisées pour rendre paramétrable — modification de la hauteur — une bibliothèque de cellules précaractérisées [BCdV91].

La méthode utilise un décompilateur qui à partir d'une structure de données physique MBK génère une description GENLIB qu'il faut rendre paramétrable. L'idée est de saisir un dessin sous un éditeur de masques symbolique et de le sauver sous une suite d'appels à des primitives GENLIB. Par exemple, un contact symbolique est directement traduisible en un appel à `PHVIA`, en passant comme arguments les informations contenues dans les champs de la structure de données. Cette description procédurale est ensuite modifiée à la main pour être rendue paramétrable en hauteur par exemple.

3.8 Générateurs utilisant des cellules précaractérisées

La vue logique doit permettre de décrire de manière simple l'instanciation de modèles, et le découpage hiérarchique. L'objectif ici est de permettre la description procédurale de *netlist*. Cette fonctionnalité pourrait être en grande partie assurée par le langage **vhdl** structurel. Cependant, la puissance du langage **vhdl** s'exprime essentiellement dans les descriptions comportementales. Dans cette optique, s'il est simple de décrire le calcul d'une racine carrée au niveau comportemental, il est beaucoup plus difficile de décrire en **vhdl** structurel un générateur paramétrable fournissant un extracteur de racine carrée en cellules précaractérisée.

Les étapes identifiées de la description d'un bloc sont :

1. définir la figure de travail, par l'intermédiaire de `DEF_LOFIG(NOM)`.
2. spécifier l'interface.
3. construire les interconnexions entre les blocs et les connecteurs de l'interface, éventuellement de manière hiérarchique. Comme pour le placement des blocs, les hiérarchies ne peuvent être imbriquées, et une figure doit être entièrement définie avant d'être utilisée.
4. sauver le résultat sur disque par `SAVE_LOFIG`. Cette fonction fait des vérifications afin de garantir la cohérence au sens de la structure de donnée MBK.

3.8.1 Fonctions de base pour définir des listes d'interconnexions

La première chose à faire lors de la construction d'un bloc consiste à définir et figer l'ordre de son interface. Cela se fait à l'aide de la fonction `LOCON(CONNECTEUR, DIRECTION, SIGNAL)`, qui crée un connecteur logique dans la figure dont la direction représente le sens de transfert de l'information, comme il a été dit figure 2.7. Le nom du connecteur est son identificateur, et doit être unique. Le signal attaché au connecteur est identifié par son nom.

L'instanciation des cellules de base ou des blocs se fait à l'aide de deux fonctions équivalentes :

`LOINS(MODÈLE, INSTANCE, LISTE_DE_SIGNAUX, 0)`
: qui ajoute une instance de la figure `MODÈLE` dans la hiérarchie courante, en affectant implicitement à chaque connecteur de la figure instanciée le signal qui a le même numéro d'ordre dans l'appel de la fonction. Cela nécessite de la part du concepteur une connaissance exacte de l'ordre de déclaration des connecteurs dans les modèles instanciés. La fonction vérifie qu'il y a autant de signaux que de connecteurs dans le modèle instancié.

`LOINSE(MODÈLE, INSTANCE, LISTE_D'ASSIGNATIONS, 0)`
: ajoute également une instance dans la hiérarchie courante, mais en explicitant complètement les relations entre signaux et connecteurs. Les assignations explicites sont de la forme `CONNECTEUR <= SIGNAL`, ce qui permet de n'avoir à connaître que le nom des connecteurs et pas leur ordonnancement. On vérifie ici qu'il y a autant de signaux que de connecteurs, et que ce nombre est égal au nombre de connecteurs du modèle instancié.

Les avantages de la version implicite sont une plus grande concision dans la description des interconnexions. En contrepartie, elle repose sur l'ordre des connecteurs dans le modèle. Elle est plus adaptée à l'utilisation d'une bibliothèque figée et documentée. La seconde version est plus verbeuse, mais plus explicite. L'ordre des connecteurs peut varier dans le modèle sans rendre la *netlist* fonctionnellement fausse. Son utilisation est plus naturelle pour l'instanciation des blocs des niveaux hauts de la hiérarchie.

Il est possible de manipuler des vecteurs de connecteurs et de signaux. Pour cela, on dispose des constructions suivantes :

`"vector[n: m]"`
: qui définit un vecteur dont les bornes sont les nombres entiers constant n et m.

`BUS("vector", p, q)`
: où les bornes p et q sont des variables entières.

`"vector[i]"`
: qui fait référence au $i^{ième}$ élément d'un vecteur, i étant un nombre entier constante.

`ELM("vector", r)`
: qui accède le $r^{ième}$ élément d'un vecteur, où r est une variable entière.

3.8.2 Fonctions de haut niveau sur la vue structurelle

Pour certain types d'applications, il est utile de pouvoir restructurer la hiérarchie. Deux fonctions complexes sont à la disposition du concepteur pour cela :

`FLATTEN_LOFIG(INSTANCE, CONCATÉNATION)`
: recopie les objets d'une instance dans le niveau hiérarchique courant et efface l'instance. Le diagramme de la figure 3.7 s'applique de la même manière ici. Cette opération est utilisée principalement lorsque l'on décrit hiérarchiquement des parties d'un bloc qui devra être routé globalement au niveau des cellules de bibliothèque.

`UNFLATTEN_LOFIG(FIGURE, INSTANCE, LISTE_D'INSTANCES, 0)`
: crée une nouvelle figure, contenant l'ensemble d'instances indiqué comme argument, et l'instancie dans la figure courante. L'interface de cette figure est obtenue à partir des noms de signaux attachés aux connecteurs des instances. Cette fonction est particulièrement utile pour découper une description en deux sous-ensembles qui

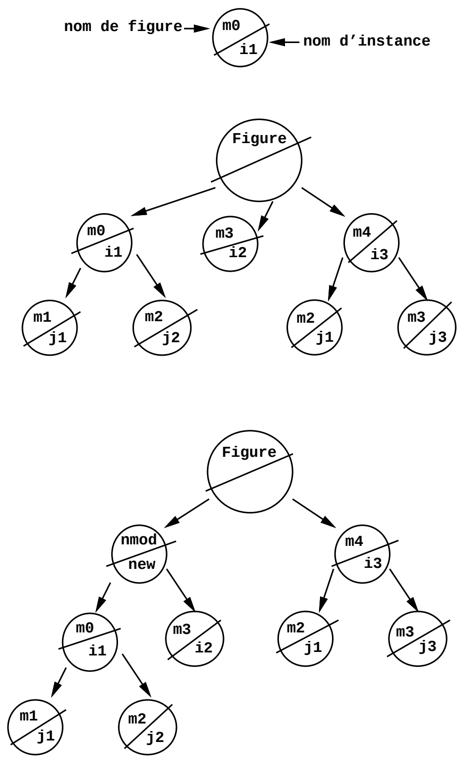


Figure 3.9: Création artificielle du niveau hiérarchique `new`, de modèle `nmod`.

ont des contraintes de routage ou de plan de masse différentes sans avoir à retoucher ni à la liste des équipotentielles, ni aux interfaces.

3.8.3 Fonctions pour le placement des cellules précaractérisées

Le placement explicite des cellules par les générateurs utilisant une bibliothèque de cellules précaractérisées permet un gain significatif en surface par rapport aux outils de placement automatique [Fra92].En contrepartie, la phase de placement n'est pas triviale et demande un temps de développement important. Sont à la disposition du concepteur :

`DEF_PHSC(FIGURE)`

ceci charge une description structurelle et crée la figure physique de nom identique.

`SC_PLACE(INSTANCE, SYMETRIE, x, y)`
`SC_TOP(INSTANCE, SYMETRIE)`
`SC_RIGHT(INSTANCE, SYMETRIE)`
`SC_BOTTOM(INSTANCE, SYMETRIE)`
`SC_LEFT(INSTANCE, SYMETRIE)`

permettent toutes de placer une cellule physique en faisant référence au nom d'instance utilisé dans la vue logique. On ne peut ainsi pas placer un modèle différent de celui qui est instancié logiquement.

`SAVE_PHSC()`

sauvegarde sur disque le placement, en effectuant des vérifications spécifiques :

- toutes les instances de la vue logique ont-elles bien été placées?

- n'y a t-il pas des cellules placées qui n'ont pas d'équivalent logique, en dehors des cellules de transparences?
- n'y a t-il pas de recouvrements topologiques entre cellules?

Le placement de cellules précaractérisées est très simple comparé à celui des blocs optimisés : toute les cellules ont la même hauteur. Elles sont placées en bandes et ne partagent que leurs connecteurs d'alimentations sur les faces EST et OUEST. Leurs connecteurs de données sont dupliqués sur les faces NORD et SUD, comme l'illustre la figure 3.10. Les interconnexions, décrites dans la vue logique, sont routées dans un canal, dont il faut tenter de minimiser la hauteur par un placement adéquat. Ici on escompte que le concepteur peut garantir des alignements dans les zones fortement connectées, et donc fournir au routeur un placement optimisé par rapport à ce que donnerait un placeur automatique.

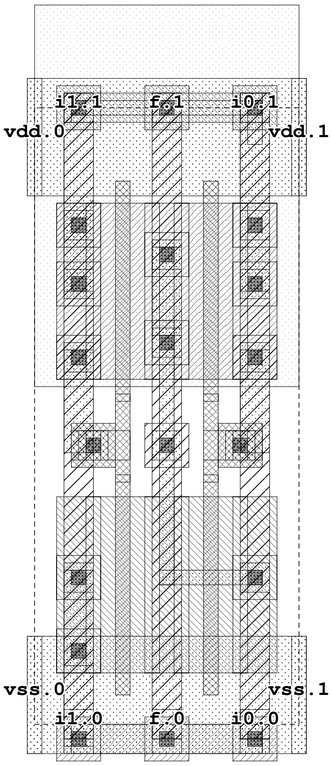


Figure 3.10: Une cellule précaractérisée typique.

3.9 Exemples de réalisations

GENLIB a été utilisé dans divers contextes.

- quatre générateurs optimisés pour chemins de données[AG93].

`rsa`, un générateur d'additionneur rapide[LGT93].

paramètres	valeurs possibles
nombre de *bits*	3 à 128
retenue d'entrée	vrai ou faux
trois entrées	vrai ou faux
drapeau de débordement	vrai ou faux

`rfg`, un générateur de banc de registres.

paramètres	valeurs possibles
nombre de *bits*	2 à 64, nécessairement pair
nombre de mots	2 à 256, nécessairement pair
nombre de bus de lecture	1 ou 2
mode d'utilisation	point mémorisant sur état ou bascule sur front
consommation réduite	vrai ou faux

`bsg`, un générateur de décaleur à barillet. Les operations possibles sont :

- décalage logique à droite.
- décalage arithmétique à droite.
- décalage logique à gauche.
- décalage arithmétique à gauche.
- rotation vers la droite.
- rotation vers la gauche.

paramètres	valeurs possibles
nombre de *bits*	3 à 64

`amg`, un générateur de multiplieur utilisant l'algorithme de Booth modifié.

paramètres	valeurs possibles
nombre de *bits* de l'opérande x	8 à 64, nécessairement pair
nombre de *bits* de l'opérande y	8 à 64, nécessairement pair
nombre d'étage de *pipeline* à insérer dans le circuit	0 à min($\frac{\mathrm{nx}}{2}$, $\frac{\mathrm{ny}}{2}$)-1

- trois générateurs de mémoires.

`grog`, un générateur de *ROM*[GP94]. Les valeurs à mettre dans la *ROM* sont spécifiées par une grammaire **vhdl**.

parametres	valeurs possibles
nombre de *bits*	1 à 64
nombre de mots	64, 128, 256, $n \times 512$, $1 \leq n \leq 8$
sortie avec commande de mise en haute impédance	vrai ou faux

`rage`, un générateur de *RAM* statique.

parametres	valeurs possibles
nombre de *bits*	2 à 128, nécessairement pair
nombre de mots	2^n, $7 \leq n \leq 12$
rapport largeur sur hauteur	étroit, moyen ou large
même bus de données en entrée et en sortie	vrai ou faux

`fifo`, un générateur de file.

parametres	valeurs possibles
nombre de *bits*	1 à 32
nombre de mots	2 à 128, nécessairement pair

- trois générateurs sur cellules précaractérisées.
  - un additionneur flottant.
  - un multiplieur flottant.
  - un convertisseur de formats.

  Pour ces derniers, deux bibliothèques de cellules ont été utilisées pour la réalisation physique, `sclib`, la bibliothèque de cellules précaractérisées de la chaîne **Alliance**, et `dplib`, la bibliothèque précaractérisées dédiée au compilateur de chemins de données.

Tableau 3.1: Données sur le développement des générateurs optimisés.

opérateur	GENLIB		**modgen**
	lignes	temps	lignes
`rsa`	1000	15 jours	11000
`rfg`	2600	4 mois	4200
`bsg`	600	1 mois	600
`amg`	1400	1.5 mois	—
`grog`	1100	1 mois	—
`rage`	1000	1.5 mois	—
`fifo`	700	1 mois	—

La table 3.1 présente le temps de développement et le nombre de lignes des générateurs optimisés. Le temps de développement ne prend pas en compte le temps de dessin des cellules. En pratique, les cellules de base sont modifiées pendant l'écriture du programme d'assemblage pour corriger les erreurs dues aux juxtapositions.

Les chiffres concernant **modgen** correspondent à une traduction manuelle du code GENLIB. En effet, dans le cadre du projet **IDPS**, trois générateurs orientés chemin de données ont été réécrits en **modgen**. Ils utilisent les mêmes cellules feuilles, seul le programme d'assemblage a été réécrit.

La taille du code de `rsa` est multiplié par 10 lors de la traduction en **modgen**, car son algorithme de placement a dû être dérécursivé, ce langage ne supportant pas ce concept. Cela a été fait de manière semi-automatique, ce qui explique partiellement ce facteur 10.

En moyenne, la compacité du code GENLIB est meilleure que celle de **modgen**, et cela s'explique par l'incapacité de décrire des tableaux de caractères à deux dimensions dans ce langage.

Ces générateurs ont été développés sur une période de trois ans, et pour la plupart par des étudiants de troisième cycle dont l'expérience en implantation de circuit était très réduite.

GENLIB est aussi utilisé pour décrire des objets uniques. Une des contraintes fondamentales de la conception *VLSI* aujourd'hui est le *time-to-market*. Dans 99% des cas, l'utilisation d'outils de synthèse sur des bibliothèques précaractérisées est la solution retenue et s'avère satisfaisante. Mais dans le 1% des cas restant, il n'est simplement pas possible de faire tenir le circuit sur le silicium ou d'avoir les performances requises avec ce genre de techniques, et il faut recourir au développement de blocs *custom*. Ces deux circuits en sont l'illustration :

- deux blocs dédiés ont été créés pour le processeur StaCS[GLWW94], démonstrateur haute complexité du projet européen **IDPS** : le premier, construit par aboutement, est un banc de registres multi-accès — 6 lectures, 4 écritures —. Il contient 50 000 transistors et a nécessité 400 lignes de GENLIB pour décrire l'assemblage. Le second est un multiplieur en cellules précaractérisées utilisant la bibliothèque standard **IDPS**. Il contient 4900 portes et a nécessité 250 lignes de GENLIB.
- GENLIB a été utilisé pour assembler un cache statique de 800.000 transistors. Ce bloc est la mémoire cache du dernier processeur DPS7 — projet Auriga II — chez BULL.

Une étude comparative entre trois générateurs de *ROM*s est présentée en annexe D. Les densités et délais des *ROM*s générées en utilisant GENLIB sont quasi-identiques à ceux obtenus pour des générateurs dédiés à une technologie — l'architecture des trois générateurs étant différente, la précision de la mesure est limitée —.

L'ensemble des fonctions de GENLIB représente environ 3600 lignes de **C**.

3.10 Conclusion

Choisir un langage de génération, c'est beaucoup plus que choisir une syntaxe, car des choix de principe du langage découle la méthodologie de conception des générateurs. Dans tous les langages de génération abordés, la base de la méthodologie de placement consiste à abouter des cellules entre elles pour former des hiérarchies, car cette approche garantit des densités élevées. Mais la manière de réaliser d'une part les cellules et d'autre part le placement est le critère déterminant dans le choix d'un langage. Comparé aux langages existants, les deux atouts majeurs de GENLIB sont la portabilité et la simplicité. Ces atouts découlent des principes du langage :

- utilisation du dessin symbolique sur grille fixe sans compaction pour les cellules de base. Cette méthode atteint effectivement ses objectifs de portabilité, comme le prouvent les résultats du chapitre 1.

  Les problèmes d'aboutement sont résolus une fois pour toutes dès que la vue symbolique est validée, car cette technique garantit la conservation des alignements quelle que soit la technologie cible. Le code d'un générateur ne contient dans ce cas que des instructions de placement d'objets, et non l'expression de contraintes d'aboutabilité. La description de l'algorithme d'assemblage et la maintenance du code en sont grandement facilitées.
- séparation claire entre le dessin des cellules de base et l'assemblage des cellules. Une cellule est avant tout une repésentation graphique. C'est sous cette forme que les concepteurs sont habitués à les voir et à les manipuler.

En contrepartie, l'assemblage d'instances consiste à placer des objets rectangulaires côte-à-côte suivant un algorithme en général relativement simple à décrire par programme, et c'est en ce sens un niveau de complexité bien moindre de celui de la description de cellule.

Nous avons prouvé, par la taille du code et le temps de développement d'un nombre important de générateurs, que cette séparation correspond à une nécessité pour la génération procédurale.

- utilisation d'un langage standard. Choisir un langage standard, c'est se donner la possibilité de compiler un générateur sur une vaste gamme d'architectures matérielles.

  L'utilisation du langage **C** a permis un développement rapide des fonctions de GENLIB. Pour l'utilisateur, c'est finalement une bonne solution également car la syntaxe de base du **C** pour les boucles, conditions, *etc*, est simple, cohérente et bien documentée. Comme le concepteur ne manipule que deux types de données à travers les fonctions de GENLIB, les notions complexes du langage lui sont cachées.

- indépendance vis-à-vis des systèmes de **CAO** commerciaux. Nous avons crée un environnement permettant de développer des bibliothèques portables sans être lié à un vendeur de **CAO** particulier, tout en garantissant l'interopérabilité avec les principaux systèmes de **CAO** actuels.

GENLIB est un langage de génération procédurale destiné à être utilisé par des concepteurs de circuits et non des programmeurs, car l'important dans un générateur est plus la qualité de la circuiterie que celle du code. Afin d'aider à l'écriture de la partie logicielle des générateurs par des non-informaticiens, il est intéressant de disposer d'outils d'aide à la mise au point. Comme le prouvent les résultats du paragraphe 3.9, l'utilisation de la vue physique symbolique sur grille fixe, du langage **C** et d'un environnement graphique dédié à l'écriture de générateurs, sujet du chapitre 4, assure une prise en main plus rapide et une meilleur productivité que les approches existantes.

Chapitre 4

Un outil de mise au point au niveau source pour l'écriture des générateurs

4.1 Introduction

GenView est un outil d'aide au développement des générateurs de circuits paramétrables obtenus par aboutement. Son but premier est de faciliter l'écriture et la mise au point de ces générateurs par des concepteurs de circuits, dont l'expérience de la programmation est *a priori* réduite, en fournissant un ensemble de fonctionnalités de mise au point sous une interface graphique.

La question posée est : Quel type d'outil permettrait d'aider un concepteur de circuit dans l'écriture et la validation d'un générateur paramétrable ou d'un bloc *custom*?

L'expérience de la programmation montre que la mise au point d'un programme est un processus itératif, et que ce qui est coûteux est aussi bien la durée de l'itération, liée au nombre et à la complexité des outils utilisé pour obtenir un résultat, que le nombre d'itérations. En effet, si l'information fournie à chaque itération n'est pas sous une forme exploitable — cette forme dépend bien entendu du problème — alors le nombre d'itérations nécessaires à l'obtention d'un programme correct sera important. Dans le cas présent, la manière la plus efficace de présenter l'information est celle qui est intuitive pour le concepteur, c'est-à-dire sous forme graphique, et afin qu'elle soit effectivement exploitable, il faut le faire parallèlement au déroulement de l'algorithme — la visualisation d'un résultat faux ne permet pas de savoir simplement d'où vient l'erreur —.

Dans le cours du développement de générateurs de circuits, la boucle de mise au point du programme est typiquement celle indiquée figure 4.1. Les exigences devant être satisfaites lors de tels développements sont :

La simplicité de visualisation
la détection et l'analyse des erreurs dans l'algorithme de construction du circuit ne peut se faire que sur le dessin. Il faut donc disposer d'un outil de visualisation de la vue physique symbolique, et ce si possible pendant le déroulement même de l'algorithme de construction.

La simplicité pour la mise au point
si les erreurs de syntaxes sont détectées par le compilateur, et donc aisément in-

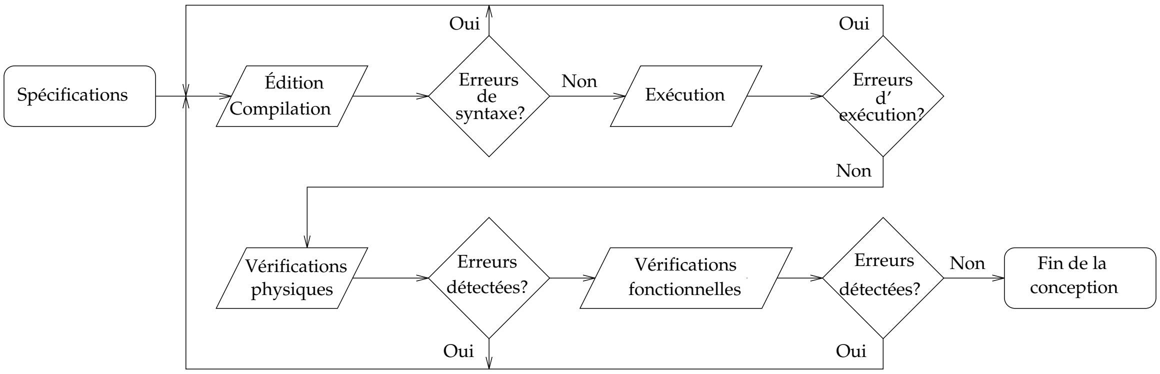


Figure 4.1: Déroulement classique de la mise au point d'un générateur.

diquées à l'utilisateur, les erreurs lors de l'exécution sont plus pernicieuses. On entend ici par erreur d'exécution non pas une erreur algorithmique, cette analyse étant du seul ressort du concepteur, mais une manipulation illicite des données par l'utilisateur. Ces erreurs systèmes sont le prix à payer pour la puissance du **C** et ses vérifications uniquement statiques. GENVIEW tente d'en simplifier la correction par la présence d'un débogueur intégré dans l'environnement. Il permet la visualisation des variables et la récupération des erreurs systèmes autorisant la reprise du travail après, par exemple, une division par zéro. GENVIEW doit prendre la main par rapport au système d'exploitation et au compilateur **C** des machines sur lesquelles il est utilisé.

La phase de vérification physique s'opère sous un éditeur graphique où seule la compétence du concepteur entre en jeu pour juger de la validité du circuit généré. Il est ici question de la détection d'erreurs algorithmiques, comme le recouvrement de deux instances, le mauvais positionnement d'un *via* dans un décodeur ..., mais pas d'une vérification fine des règles de dessin. La validation fonctionnelle est beaucoup plus complexe, et suppose l'utilisation des outils de vérification imposés par la méthodologie — extracteurs, simulateurs — qui sortent du cadre de cette discussion.

4.2 Les outils de mise au point : état de l'art

Les langages interprétés, comme le **lisp**, permettent à tout instant l'arrêt du programme pour consulter l'état des variables, ce qui facilite la mise au point du code.
Les langages compilés donnent des exécutables beaucoup plus rapides et permettent des optimisations importantes du code. Mais l'action interactive sur le code n'est plus possible. Il faut donc des outils nouveaux qui permettent la mise au point *symbolique*, c'est-à-dire corrélée au code source, des exécutables.

4.2.1 Les *debugger*s classiques

Tous ces outils ne fonctionnent symboliquement que si les exécutables ont été compilés avec une option permettant de conserver la table des symboles utilisée à la compilation.

Deux *debugger*s sont maintenant disponibles en standard avec UNIX. **Dbx** et **gdb** fournissent à peu près le même genre de services, et possèdent chacun une interface graphique permettant une relative souplesse pour la lecture du code source, le choix des variables à examiner, *etc*, pour **fortran**, **C**, **C++**. Ce sont des outils très complets, avec de grandes capacités symboliques. Les conversions de types sont automatiques, et on peut afficher des types composites, tableaux et structures. L'utilisation de la ligne de commande est un peu ardue, mais très puissante. On peut par exemple spécifier des conditions sur la trace de variables ou l'arrêt du programme.

Les interfaces graphiques, comme celle de la figure 4.2, facilitent l'utilisation en affichant le code source sur lequel on peut poser des points d'arrêts, indiquer des noms de variables, *etc*, mais en ne permettant qu'une petite partie des actions autrement disponibles. Néanmoins l'expérience montre que le programmeur moyen aura beaucoup plus tendance à utiliser le mode graphique, car les commandes des menus sont suffisantes dans la plupart des cas.

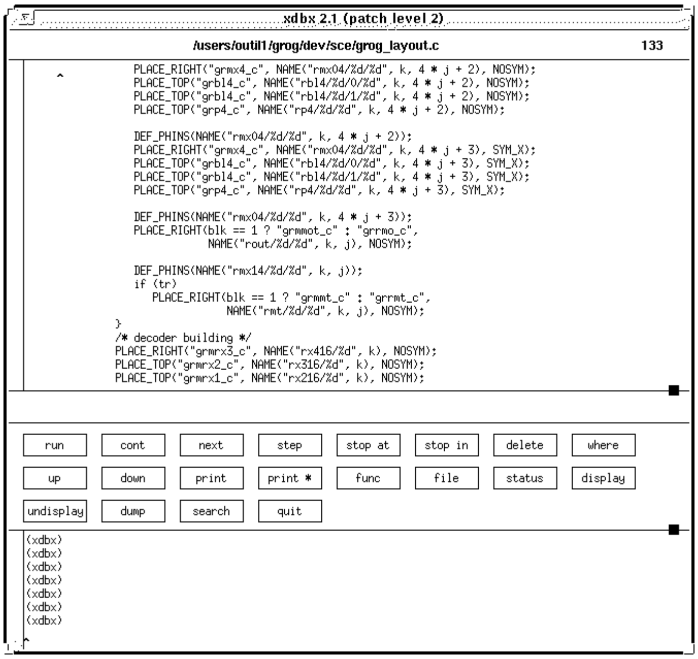


Figure 4.2: L'interface graphique de l'outil **dbx**.

Si ces programmes sont performants pour mettre au point du code **C** général, ils exigent d'importantes notions de programmation et ne sont pas adaptés à la spécificité de la génération de circuits. Dans la description procédurale du dessin d'un circuit, l'interprétation du contenu de variables n'est pas intuitive, d'autant plus que le placement est généralement relatif. Seule une approche graphique peut fournir le niveau d'interaction désiré dans le contexte du placement par aboutement.

4.2.2 Les *debugger*s orientés génération procédurale

Un outil de mise au point de code dédié à la génération procédurale est l'interpréteur du langage **Skill**[WL86], intégré dans CADENCE. Ce n'est pas à proprement parler un outil de mise au point spécifique pour les générateurs, mais puisque le langage est destiné à la gestion de l'ensemble des outils de cette chaîne, il est possible de lui adjoindre un *debugger*. Un des effets direct est de permettre une mise au point des programmes de placement, en permettant de visualiser les phases de construction.

Sous **Skill**, l'entrée dans le mode *debug* se fait soit par le positionnement d'un point d'arrêt sur une définition de fonction, soit parce que l'interpréteur a détecté une erreur à l'exécution. Il faut néanmoins avoir préalablement installé le *debugger* comme fonction de récupération d'erreur. Une fois dans ce mode, il est possible d'examiner l'état du programme, la valeur des variables locales, l'ordre d'appel des fonctions, *etc*. L'exécution en pas à pas est possible mais il faut l'utiliser avec prudence car le comportement sur une fonction système est indéfini.

Il est possible de tracer l'ordre d'exécution des fonctions et la valeur de leurs paramètres et valeurs de retour, à l'aide de fonctions spécialisées. La trace de variables, locales ou globales s'effectue avec les commandes **Skill** usuelles, ce qui permet de faire des actions complexes.

Il manque à cet outil la souplesse d'un environnement graphique du type de **xdbx**, car toutes les commandes doivent être saisies sur une unique ligne dans la fenêtre **CIW**, la fenêtre de gestion d'OPUS.

Une autre tentative dans le même esprit, c'est-à-dire langage **lisp** plus dialogue sur une ligne de commande, a donné l'outil **MADMACS**[GP92] développé à l'IRISA. Combinant une action graphique à l'exécution de commandes de dessin basé sur un interpréteur **lisp**, l'utilisateur a à sa disposition un sous ensemble du **lisp** et des fonctions prédéfinies pour déplacer un curseur utilisé comme référence pour le placement. De même que pour **Skill**, la gestion des actions de mise au point se fait textuellement, ce qui limite la qualité de l'interaction avec l'utilisateur.

4.2.3 Critique des outils existants

Les outils de mise au point des langages usuels ne sont pas adaptés à la spécificité du problème. Les algorithmes de placements dans les générateurs sont des algorithmes de dessins, et l'interprétation de la valeur de variables ne renseigne pas explicitement sur leur validité. De plus, le placement est essentiellement relatif et les seules variables utilisées pour le codage de l'algorithme concernent les noms des objets ou les variables d'itération et de contrôle, mais pas les coordonnées — même si c'était le cas, ce ne serait pas simple à interpréter —.

Les — rares — outils dédiés exclusivement à la mise au point de programme de génération procédurale ont les défauts suivants :

- ils manquent d'interactivité. Les opérations de mise au point sont saisies sur une ligne de commande, ce qui est en contradiction relative avec l'aspect graphique lié au dessin.
- l'édition et l'interprétation ne se font pas physiquement au même endroit, alors que ces deux actions sont très liées et qu'elles ont un impact direct sur le texte source.
- ils sont dédiés à l'interprétation de langages spécialisés interprétés, ce qui va à l'encontre du besoin de portabilité logicielle des générateurs.

L'étude précédente nous permet de constater que :

1. les fonctions avancées des outils généraux de *debug* sont très rarement utilisées. Elles sont inutiles dans le cadre de la génération procédurale.
2. l'aspect graphique est essentiel dans le contexte *VLSI*.

Afin de satisfaire les besoins de simplicité, d'interaction et de portabilité, l'outil GENVIEW a été développé.

4.3 Architecture globale de l'outil

GENVIEW est scindé en trois pôles principaux, correspondant aux trois phases de développement d'un générateur paramétrable vu de l'utilisateur.

Édition
: Une description programmée de circuits dans l'environnement MBK passe par l'écriture d'un programme source contenant des appels aux fonctions de la bibliothèque GENLIB. Il faut donc avoir la possibilité d'éditer des fichiers, pour en visualiser le contenu, le modifier et le sauver. L'édition est évoquée paragraphe 4.4.

Exécution
: Une fois le fichier saisi, il faut l'exécuter afin qu'il produise un résultat. Pour cela, il faut d'abord compiler le fichier source en un fichier objet, puis faire l'édition de lien pour rendre accessible les bibliothèques externes, et enfin l'exécuter. Cette partie est détaillée en 4.5.

Visualisation
: Finalement, il faut pouvoir visualiser les objets créés, afin d'évaluer la justesse des algorithmes mis en œuvre. Cette phase est décrite en 4.6.

GENVIEW permet tout cela sous un unique environnement graphique multi-fenêtres. L'outil utilise le système de fenêtrage *X Window System Version 11*[Nye88], qui donne la possibilité de créer des applications graphiques portables. L'outil contient trois fenêtres principales, comme on peut le voir figure 4.3. Chacune possède des menus adaptés qui permettent l'édition du texte, la visualisation des erreurs rendues par le système, et la visualisation graphique de la figure en construction. Du point de vue de l'architecture interne, GENVIEW est proche de la boucle de conception évoquée précédemment, et la communication entre ses diverses parties est matérialisée figure 4.4.

L'objectif principal de GENVIEW est de réduire le temps de développement. L'accent, d'un point de vue informatique, a été mis sur la performance. Les goulots d'étranglement sont :

1. le temps de compilation et d'assemblage.
2. le temps de l'édition de liens.
3. le temps d'exécution.
4. le temps de visualisation.

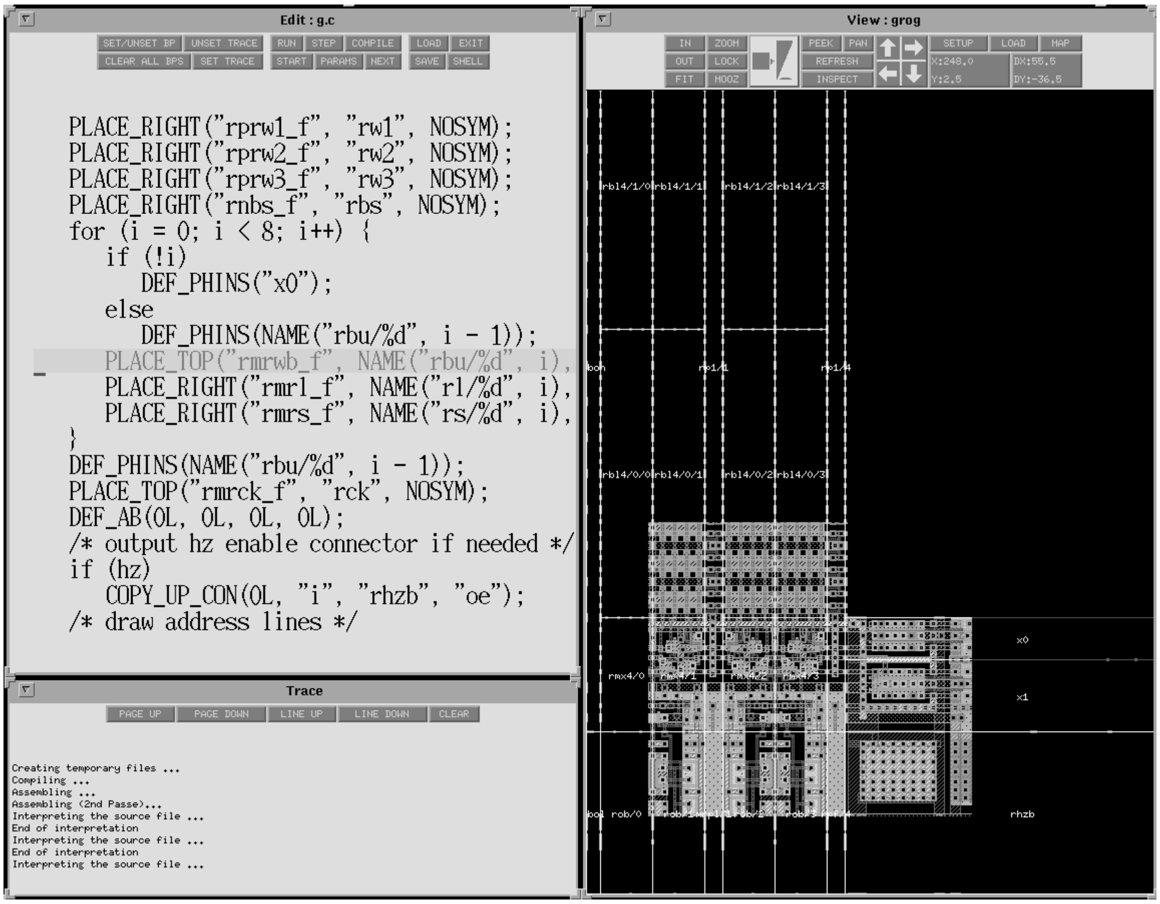


Figure 4.3: Utilisation pratique de l'environnement graphique GENVIEW.

Le point 1 est incompressible. Sous GENVIEW, le point 2 est inutile, car c'est lors de l'interprétation que l'on va vérifier la présence des fonctions utilisateur. La phase d'obtention d'un code interprétable sera sensiblement plus rapide sous GENVIEW que sous l'environnement standard, car l'édition de liens avec les librairies MBK et GENLIB est coûteuse — de l'ordre de la trentaine de secondes sur une **Sparc I** —.
Le point 3 est délicat, car un interpréteur, aussi austère soit-il, est une couche supplémentaire lors de l'exécution. C'est aussi pendant cette phase que l'interaction avec l'utilisateur est la plus forte car l'affichage se fait en parallèle. Pour la visualisation, point 4, l'effort a porté sur la réduction des temps de réponse dans les opérations de zoom et de déplacement.

4.4 L'éditeur de texte

Une des contraintes de l'éditeur de texte de GENVIEW est de pouvoir communiquer avec l'interpréteur. La communication de l'interpréteur vers l'éditeur se fait à travers une fonction qui saute à une ligne et la met en inverse vidéo, ce qui permet de suivre l'exécution du programme ligne à ligne. Dans l'autre sens, l'information nécessaire est la connaissance des points d'arrêts pour la gestion du déroulement de l'exécution. Elle est fournie à l'in-

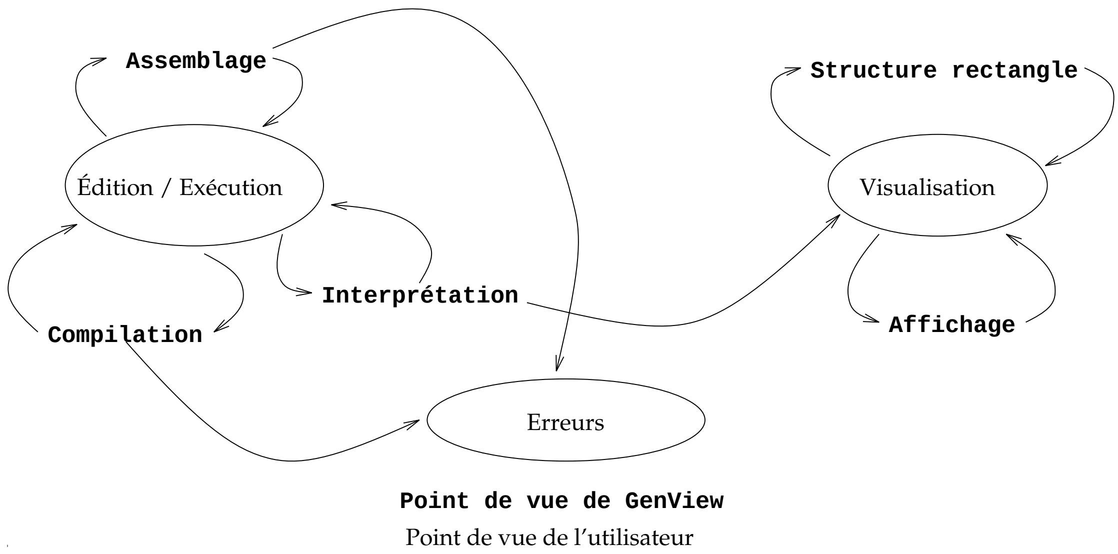


Figure 4.4: Architecture modulaire de l'outil.

terpréteur sous la forme d'une liste de numéro de lignes sur lesquels il faut stopper.

Aucun des éditeurs standards ne permet une telle interaction avec le monde extérieur, il a donc fallu développer un éditeur de texte relativement complet en mesure de satisfaire les concepteurs. Pour cela, il a été choisi d'utiliser un sous-ensemble des commandes de `vi`, l'éditeur sans doute le plus utilisé sous UNIX, auquel les utilisateurs de GENVIEW ont tous été confrontés.

4.5 L'interpréteur

4.5.1 Choix du mécanisme d'interprétation

Le langage **C** a été choisi comme support à GENLIB pour les raisons exposées au chapitre 3. Trois possibilités sont offertes pour exécuter le code :

- soit compiler totalement en mémoire vers un arbre d'exécution. Il n'y a pas de code intermédiaire ici, et c'est en fait une interprétation directe du **C**.
- soit compiler vers un code intermédiaire en langage d'assemblage et interpréter ce code symbolique.
- soit compiler vers un code intermédiaire en langage d'assemblage, assembler le code intermédiaire et interpréter le code binaire.

Chaque mécanisme a ses avantages et ses inconvénients, et le choix dépend des objectifs qui sont :

- interprétation de l'ensemble des constructions du langage **C**. Comme le langage GENLIB est un ensemble de fonctions **C**, il n'y a pas de raisons d'empêcher le concepteur de créer ses propres fonctions et ses propres structures.

- exécution atomique des fonctions de la bibliothèques GENLIB. Il n'est pas souhaitable de permettre d'exécuter en pas à pas le corps des fonctions de bibliothèques, car elles sont vues comme une action unique par le concepteur.
- unicité des sources des fonctions de bibliothèque. Du point de vue de la maintenance et de l'évolution de l'outil, il est nécessaire de ne pas avoir à dupliquer les sources des fonctions de bibliothèque, fonctions GENLIB ou fonctions de la bibliothèque standard du **C**, dans le code source de l'interpréteur.
- disponibilité des fonctions de *debug*. Le but de l'outil étant la mise au point d'un programme de génération, il faut permettre l'exécution en pas à pas, la trace des variables, *etc*.
- obtention d'un temps d'itération court. Le temps de développement de l'algorithme d'assemblage est fortement lié à la durée de la séquence modification, compilation, exécution, vérification . Il faut s'attacher à rendre ce temps aussi court que possible.
- minimisation du coût du développement de l'interpréteur. Il faut en limiter le temps d'implémentation, car il ne représente qu'une partie de l'application visée.

Les trois solutions imposent l'utilisation d'un compilateur. La solution la plus économique pour le développement consiste à utiliser un compilateur paramétrable existant. Ceci écarte de fait la première solution et impose par la même définition d'une machine virtuelle. Le choix entre les deux autres alternatives repose sur une évaluation de la complexité de l'interpréteur. Si l'on n'assemble pas, il faut gérer l'allocation des ressources — variables, paramètres — dynamiquement. De plus, il faut vérifier l'existence des fonctions non résolues dans le code à chaque appel, car elles peuvent être déclarées dans la suite. Le modèle d'interprétation est plus compliqué dans ce cas. En assemblant vers un code binaire, les ressources sont directement allouées dans la zone mémoire de données, et les fonctions externes sont identifiées. Ce modèle de programme est beaucoup plus simple pour l'interpréteur. On peut espérer un gain de temps aussi bien pour le développement, car les assembleurs sont des outils bien connus, que pour l'exécution, car la phase d'assemblage est rapide et fournit un code interprétable très régulier.

En conséquence, le mécanisme choisi est le suivant :

1. compiler avec un compilateur **C** paramétrable vers une architecture la plus simple possible, pour obtenir un fichier en langage d'assemblage.
2. assembler les instructions assembleurs dans le langage machine de la machine virtuelle vers un tableau en mémoire.
3. exécuter le langage machine de la machine virtuelle en faisant dynamiquement l'édition de lien pour les fonctions non résolues.

Si les deux premiers points sont clairs, le troisième mérite quelques explications.

Les fonctions non résolues sont les fonctions qui ne sont pas décrites dans le fichier source. Ce sont soit des fonctions de GENLIB soit des fonctions de la bibliothèque standard du **C**. On pourrait imaginer deux manières de rendre ces fonctions accessibles :

- précompiler les fonctions de bibliothèques dans le langage de la machine virtuelle. Dans ce cas, lors de l'édition de lien à la fin de l'assemblage, les fonctions sont insérées dans le code binaire de la machine virtuelle. Les points positifs de cette approche sont :

  – la totalité du code est alors interprétable par la machine virtuelle, ce qui conceptuellement est un modèle simple. C'est d'ailleurs ce qu'il se passe sur les machines réelles.

  – la récupération des paramètres par les fonctions prédéfinies est prise en compte par le compilateur.

  En contrepartie, ses inconvénients en font une technique difficilement envisageable, car :

  – il faut posséder les sources des fonctions de bibliothèques. C'est finalement un problème très complexe, car dans le corps même des fonctions de GENLIB sont utilisés une multitude de fonctions, ainsi que des appels systèmes. De plus, les fonctions de placement doivent appeler des routines graphiques pour l'affichage de la figure en construction. On imagine mal la totalité de l'environnement graphique s'exécutant sur la machine virtuelle.

  – en imaginant que l'on puisse obtenir toutes ces sources, le temps d'interprétation serait à coup sûr assez élevé, car un interpréteur aussi simple soit il est pénalisant par rapport à l'exécution sur une machine réelle.

  – il faut gérer explicitement l'exécution atomique des fonctions de bibliothèque, car c'est l'une des fonctionnalités désirées de l'outil.

- utiliser les fonctions de bibliothèques compilées sur la machine hôte. Les fonctions de bibliothèque sont incluses dans le corps de l'interpréteur, de manière quasi-similaire à n'importe quelle instruction de la machine virtuelle. C'est un mécanisme d' interprétation partielle , qui n'interprète avec les instructions de la machine virtuelle que le programme utilisateur, et qui exécute sur la machine réelle les fonctions de bibliothèque. Les avantages sont :

  – une grande simplicité de mise à disposition de nouvelles fonctions de bibliothèque. Comme ici le code de la fonction est compilé sur la machine hôte, il n'est pas nécessaire de posséder les sources des fonctions appelées. Seule l'interface de la fonction est nécessaire.

  – l'exécution atomique des fonctions de bibliothèque est garantie par construction, puisqu'elles sont implémentées quasiment comme les instructions de la machine virtuelle.

  – la rapidité d'interprétation est supérieure à celle de l'approche précédente, car la vitesse d'exécution des fonctions de bibliothèque est celle de la machine hôte, réelle, et non celle de la machine virtuelle.

  Le seul inconvénient est :

  – la nécessité d'encapsuler les fonctions de bibliothèque. En effet, il faut faire la conversion des conventions de passage de paramètres entre la machine virtuelle et la machine hôte, ainsi que la conversion de la valeur de retour des fonctions de convention de la machine hôte en la convention de la machine virtuelle. Si cela ne pose pas de problèmes en ce qui concerne l'interface des fonctions à nombre de paramètres fixe, les fonctions à nombre variable de paramètres nécessitent un traitement particulier.

Il faut de toute façon encapsuler les fonctions de GENLIB qui produisent des objets affichables car on désire conserver l'unicité des sources tout en effectuant des actions graphiques.

C'est clairement le mécanisme d' interprétation partielle qui est le plus simple à mettre en œuvre et le plus performant.

4.5.2 Le compilateur

Nous avons choisi le compilateur de la *Free Software Fundation*, un organisme universitaire américain qui propose un compilateur **C** nommé *gnu cc* entièrement paramétrable à travers quatre fichiers de configuration. Afin de simplifier cette paramétrisation, la machine virtuelle doit être facile à décrire pour **gcc** et le code généré doit être simple à interpréter sous GENVIEW — ces deux objectifs sont d'ailleurs assez proches —.

Deux solutions sont possibles pour choisir une machine cible :

- utiliser une architecture existante, dont le fichier de configuration est fourni avec la distribution de **gcc**, et interpréter le code de cette machine.
- définir une architecture dédiée, la spécifier et créer les fichiers de configuration du compilateur la décrivant.

Le premier point permet de récupérer un compilateur sans aucun effort. En contrepartie, l'assembleur et l'interpréteur sont plus délicats à mettre en œuvre. Les machines cibles standards du compilateur sont des architectures réelles, donc complexes, **RISC**, comme le *Sparc* ou le *Mips*, ou **CISC**, comme le *68000* ou le *386*.
Le second permet de définir l'architecture cible la plus régulière et la plus simple possible, afin de faciliter l'assemblage et l'interprétation. Mais pour cela, il faut définir une architecture, et la coder en langage **RTL**[S+89].

La seconde solution a été choisie, car au début du projet, la partie complexe pressentie était l'interprétation du code binaire généré par l'assembleur. (*A posteriori*, la description d'une machine sous **gcc** s'avère relativement complexe.) Pour simplifier l'écriture de l'interpréteur, on peut définir une machine très régulière, possédant des instructions permettant une représentation de type code à trois adresses , qui prend en mémoire la forme d'un graphe orienté acyclique[ASU86]. Leur forme générale est `x = y` *op* `z`, ou *op* est un opérateur quelconque du langage **C**. Cette terminologie se justifie car chaque instruction possède usuellement des références à trois adresses, deux pour les opérandes, une pour le résultat.

L'annexe F détaille la méthode de paramétrisation du compilateur et présente le jeu d'instructions de la machine.

4.5.3 L'assembleur et le code produit

La représentation binaire interprétable est très simple. Les instructions sont codées sur 32 *bits*, en quatre octets d'utilité distincte. Le format de l'instruction est :

code opération	mode opérande 0	mode opérande 1	mode opérande 2
8 bits	8 bits	8 bits	8 bits

Le premier octet du code opération contient un code pour l'instruction et les trois autres le mode d'adressage de chacun des éventuels opérandes. Le codage ne se veut pas compact mais simple à interpréter. Le format du mode d'adressage est :

Mode d'adressage			Numéro de registre				

Les trois *bits* de poids fort codent le mode d'adressage de l'opérande considéré. Il y a cinq modes d'adressage valides, indiqués paragraphe F.3. Lorsqu'un registre est impliqué dans une opération, alors les cinq *bits* de poids faible de l'instruction codent le numéro du registre. Ils sont inutilisés pour les modes n'utilisant pas les registres, et les adresses ou constantes sont données comme opérandes sur 32 *bits* dans les mots suivant l'instruction. Les instructions sont donc de longueur variable, de 1 à 4 mots de 32 *bits*.

L'assemblage se fait classiquement en deux passes. La première passe lit le source assembleur et le convertit en un binaire compréhensible par la machine virtuelle, en laissant non résolus les symboles, c'est-à-dire les références aux données, et les étiquettes, soit les références au code. Une table des symboles est créée, associant à chaque symbole ou étiquette une valeur unique. Les directives de compilations qui indiquent la correspondance entre une ligne de source **C** et un paquet de mots sont converties en instructions qui n'agissent pas sur le déroulement du programme, mais qui informent l'éditeur de texte de la ligne du source en cours d'exécution. Durant la phase de production du binaire, les informations pour la mise au point concernant le nom, le type et la localisation des variables du programme à tracer sont analysées et conservées dans une structure de données appropriée. La structure est destinée à être lue par l'outil de mise au point lors de la trace de variables au cours de l'exécution. Ces pseudo-instructions ajoutées pour la mise au point sont détaillées dans l'annexe F.5.

Du point de vue de l'implémentation, l'analyseur du fichier assembleur utilise les outils **lex**[Les75] et **yacc**[Joh75], et l'analyseur des pseudo-instructions utilise **yacc**, qui simplifie grandement l'analyse des grammaires récursives.

La seconde passe résout les références inconnues en leur substituant la valeur calculée à la fin de l'analyse du fichier, et remplace les étiquettes inconnues, c'est-à-dire les appels à des fonctions qui ne sont pas décrites dans le fichier source, en substituant à l'appel de fonction qui les précède une interruption, instruction inconnue du compilateur mais connue de l'interpréteur, qui prend comme argument la valeur associée à l'étiquette dans la table des symboles. (Il n'est pas possible d'avoir des symboles non résolus en ce qui concerne les données, car on suppose l'utilisation d'un fichier unique pour le générateur, donc sans référence à des variables externes.) Le but de cette instruction est d'indiquer clairement que l'action à effectuer doit être recherchée dans la librairie standard de l'interpréteur.

L'assemblage fournit comme résultat deux tableaux, l'un contenant le code sous forme binaire directement interprétable, l'autre les données statiques et globales auxquelles il est fait référence dans le code, et une structure chaînée nécessaire au suivi des variables. Cette liste contient les adresses dans le code des fonctions, les déplacements par rapport aux adresses des fonctions des paramètres et des variables locales, et les adresses dans les données des variables statiques et globales. Elle utilise la table des symboles construite par l'assembleur pour la seconde passe.

4.5.4 L'interpréteur

Le programme à interpréter se trouve sous la forme d'un tableau binaire contigu en mémoire, qui contient les instructions et leurs opérandes. L'interpréteur va commencer l'exécution du code à l'adresse de la fonction `_main`, qui est le point d'entrée de tout programme **C**.

4.5.4.1 Interprétation des opérateurs

L'interpréteur doit pouvoir gérer des types d'instructions différentes. À chaque valeur de code opération correspond une fonction **C** dont les opérandes vont être lus dans le code binaire lors de l'appel en tenant compte du mode d'adressage. Une fois les modes d'adressages et les adresses, constantes ou registres, connus, l'opération est effectuée. Ces opérations sont de plusieurs types :

Les instructions issues de la compilation
Ce sont les instructions de la machine virtuelle. Elles agissent sur le comportement de la machine virtuelle en altérant la valeur de ses registres généraux, de son compteur ordinal, de la mémoire, en un mot, décrivent l'algorithme codé.

L'instruction `line`
Celle-ci est l'interface entre le binaire et l'éditeur de texte. Son argument est un numéro de ligne, et l'éditeur centre sa fenêtre de visualisation sur cette ligne lors de l'appel. Son impact sur le déroulement de l'algorithme est nul.

L'instruction `irq`
Lors de l'exécution d'une instruction `irq`, l'interpréteur va récupérer le nom de la fonction associée à son argument dans la table des symboles de l'assembleur. Si cette fonction est connue, alors, c'est qu'il s'agit d'une des fonctions de la bibliothèque implémentée dans GENVIEW. Dans le cas contraire, c'est que la fonction n'est pas définie, et GENVIEW en informera l'utilisateur.

4.5.4.2 Encapsulation des fonctions externes

GENVIEW à pour but de permettre l'interprétation du langage GENLIB. Pour cela, il faut que l'interpréteur connaisse le nom de chacune des fonctions GENLIB qui doivent être exécutées de façon atomique. De plus, certaines fonctions standards du système sont utiles pour le développement, comme la famille `printf`, qui permet d'afficher des messages sur le terminal, ou la famille `str`, qui permet de manipuler des chaînes de caractères.

Une interface spécifique est crée dans GENVIEW pour pouvoir appeler chacune des fonctions externes, qu'elles soient de la bibliothèque standard ou de GENLIB. À chaque

fonction externe prédéfinie correspond une fonction précompilée qui est intégrée dans le code lors de l'édition de lien et qui permet d'encapsuler la fonction externe. Pour exécuter une telle fonction, on appelle la fonction précompilée qui a été incluse dans le code lors de l'édition de lien à la compilation de GENVIEW. Il faut donc passer les paramètres de l'utilisateur, qui sont présents dans le binaire de la machine virtuelle que GENVIEW interprète, à la fonction **C** précompilée sur la machine hôte. On va lire les paramètres dans la pile d'exécution, car l'on maîtrise la méthode utilisée par le compilateur/interpréteur pour passer les arguments des fonctions. Ces arguments vont être mis dans des variables intermédiaires, elles mêmes passées comme paramètres à la fonction réelle. Ici, c'est la méthode de passage de paramètres du compilateur de la machine sur laquelle GENVIEW est exécuté qui va être utilisée. La valeur de retour de la fonction, quand elle est censée être utilisée est stockée dans le registre `d25`, qui a été indiqué au compilateur comme contenant la valeur de retour des fonctions.
Un exemple d'encapsulation de fonction est fourni ci-dessous, pour la fonction `GET_CON_X` de GENLIB.

```
void
   call_to_GET_CON_X()
{
char *insname;
char *refname;
char face;

   insname = get_parameter(0, PTR).ptr;
   refname = get_parameter(4, PTR).ptr;
   face = get_parameter(8, INT).int;
   reg[25] = GET_CON_X(insname, refname, face);
}
```

La fonction `GET_CON_X` est la fonction précompilée. La fonction `call_to_GET_CON_X` est une fonction sans paramètre qui est exécutée lors de l'interruption logicielle. La fonction `get_parameter` va lire une donnée dans la pile, à la profondeur spécifiée par son premier paramètre et du type de son second. Les types attendus sont calqués sur les classes que génère le compilateur pour les paramètres, soit :

`INT` pour les types `char`, `short`, `int` et `long`. On remarque que les paramètres sont tous alignés sur une frontière de mot, soit 4 octets.

`REAL` pour les types `float` et `double`. La norme du **C** spécifie en effet la transformation des flottants simple précision en flottant double précision lors du passage de paramètres.

`PTR` pour tous les types de pointeurs.

Le schéma de la figure 4.5 illustre les actions associées à l'appel dans la machine virtuelle d'une fonction de bibliothèque.

Le seul problème de cette approche concerne les fonctions à nombre de paramètres variables, car on ne sait passer qu'un nombre fixe de paramètres à la fonction encapsulée. La seule solution trouvée manque d'élégance, et consiste à limiter arbitrairement le nombre maximal d'arguments et à toujours les passer tous à la fonction encapsulée. Les paramètres non significatifs sont simplement ignorés par la fonction appelée.

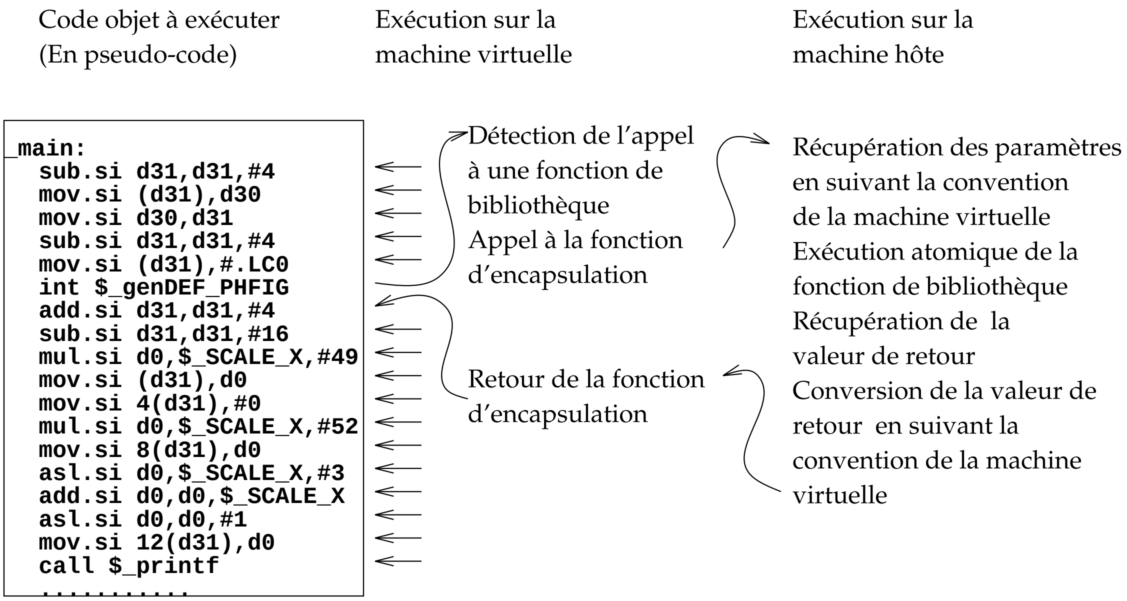


Figure 4.5: Mécanisme d' interprétation partielle .

4.5.4.3 Actions graphiques liées à l'interprétation

Un des buts de GENVIEW étant de permettre de visualiser un circuit en cours de placement, l'interprétation va devoir fournir des ordres graphiques.

Chaque fonction de GENLIB renvoie par construction un pointeur sur l'ensemble des objets crées.

La communication entre la visualisation et l'interpréteur se fait par l'intermédiaire de fonctions spécialisées qui prennent en entrée un pointeur sur un objet de la structure de données MBK, le transforment en un ensemble de rectangles et l'affichent. Chacune des fonctions de GENLIB étant encapsulée pour l'appel comme présenté en 4.5.4.1, il n'y a aucune difficulté à ajouter un appel à la fonction de création de rectangles. L'unique fonction de la visualisation connue de l'interpréteur est `fill_descriptor`, qui prend comme premier argument un pointeur sur un objet MBK, et comme second argument le type de cet objet. Ainsi, on construit la structure de données rectangles définie paragraphe 4.6.3, qui peut être lue et affichée directement. Un exemple de fonction créant un objet affichable est fourni ci-dessous pour la fonction `PLACE` de GENLIB:

```
void
   call_to_PLACE()
{
char *figname, *insname;
long x, y;
char sym;

   figname = get_parameter(0, PTR).ptr;
   insname = get_parameter(4, PTR).ptr;
   sym = get_parameter(8, INT).int;
   x = get_parameter(12, INT).int;
   y = get_parameter(16, INT).int;

   fill_descriptor(PLACE(figname, insname, sym, x, y), INSTANCE);
```

```
}
```

Ou la fonction `PLACE` retourne un pointeur sur l'objet de type instance crée.

4.5.4.4 Actions de l'utilisateur sur l'interprétation

Le fonctionnement de l'interpréteur est soumis aux décisions de l'utilisateur quand à la manière de faire s'exécuter le programme.

Trois modes sont disponibles :

1. le mode `RUN`, qui exécute la totalité du programme. Néanmoins, sous l'éditeur l'utilisateur peut demander la pose de points d'arrêt qui font que lors de l'arrivée à la ligne de code spécifiée, l'exécution s'arrête. Dans cette configuration, les actions graphiques ne sont répercutées sur l'affichage que lors de l'arrivée sur un point d'arrêt ou à la fin du programme.
2. le mode `STEP`, qui exécute ligne par ligne, et qui entre dans les fonctions utilisateur. Ici, la ligne en cours d'exécution est mise en surbrillance et centrée sous l'éditeur. Les objets générés sont immédiatement dessinés dans la fenêtre de visualisation.
3. le mode `NEXT`, qui exécute ligne par ligne mais qui exécute les fonctions utilisateur de façon atomique. Ainsi, une fonction dont le comportement est validé ne sera pas parcourue chaque fois qu'elle est appelée.

4.5.5 L'outil de trace

Cet outil est intégré dans l'interpréteur car il y est intrinsèquement lié. Son but est de pouvoir, à chaque instant, donner la valeur des variables du programme en cours d'exécution.

Suivant le genre de la variable, paramètre, variable locale, automatique statique ou globale, l'utilisation de l'adresse fournie lors de l'assemblage sera différente.

paramètre l'adresse est en fait un déplacement dans la pile par rapport à l'adresse dans la pile où a été empilée la fonction dans laquelle le paramètre est défini.

variable locale automatique l'adresse est un déplacement par rapport à l'adresse de la fonction déclarant la variable locale. Ce déplacement sera utilisé de manière identique aux paramètres.

variable statique ou globale l'adresse est l'adresse absolue par rapport au début des données.

La difficulté principale est d'être capable d'afficher tous les types de variables, même ceux que l'utilisateur a été amené à définir pour simplifier ses algorithmes.

La seule possibilité offerte est de pouvoir suivre l'évolution de la valeur d'une variable, d'un tableau ou d'une structure. Il n'est pas possible, à la différence des outils de mise au point du langage **C**, d'utiliser des constructions syntaxiques pour explorer les structures de données.

4.5.6 Récupération des erreurs

Sous l'environnement GENVIEW, deux types d'erreurs graves peuvent avoir lieu :

- les erreurs de programmation liées au système, comme par exemple dépasser les limites d'un tableau **C**, ou diviser un nombre par zéro, *etc*. Lors de telles actions, le système se défend en envoyant au processus fautif un *signal* indiquant la nature du problème, qui a normalement pour effet de l'interrompre.
- les erreurs liées à l'exécution du générateur. Le langage GENLIB est défini de telle sorte qu'il n'y ait pas la possibilité de récupérer les erreurs en phase d'exploitation, quand par exemple il est impossible de trouver les cellules de base d'un bloc. Mais c'est évidemment impensable dans le cadre d'un outil de mise au point. Pour ce faire, il a été décidé que les fonctions MBK échouant appellent une fonction `EXIT`, qui en fait va tuer le processus en lui lançant un signal de terminaison. Ceci permet de considérer de manière similaire les problèmes utilisateur et système.

La gestion des signaux du système d'exploitation permet de récupérer la main dans un état connu, et donc d'informer l'utilisateur de ses erreurs, en indiquant la nature du problème et de la ligne de code incriminée. Cette approche a le désavantage de rendre GENVIEW dépendant du système d'exploitation UNIX.

4.6 L'interface graphique

Les aspects graphiques d'une telle application sont déterminants, car ils définissent l'ergonomie de l'outil.

Pour être attractif, un outil graphique doit être :

- rapide pour afficher les données attendues. Les temps de latence liés à l'interprétation sont traités au paragraphe 4.3. C'est ici le côté affichage qui doit être optimisé.
- appréhensible aisément, sans avoir recours à une documentation compliquée.
- esthétiquement attirant.

4.6.1 Interface homme-machine

Le but ici est d'avoir une interface intuitive et simple à utiliser. Pour cela, il faut[Mar83] :

1. séparer clairement les fonctionnalités de chaque fenêtre.
2. une cohérence globale au niveau de l'aspect des menus.
3. avoir une uniformité de comportement de la souris.
4. utiliser des couleurs parlantes pour le concepteur de circuits *VLSI* pour la représentation des niveaux technologiques.

Le point 1 fait apparaître essentiellement trois fenêtres : la fenêtre d'édition, la fenêtre de visualisation du dessin et la fenêtre de trace.

L'interaction effective entre l'édition et l'interprétation est assez faible, puisqu'elle ne concerne que le positionnement des points d'arrêt et le défilement du texte pour la mise au point. Utiliser une unique fenêtre est néanmoins naturel, car les deux fonctions manipulent le même matériau qui est le texte source du générateur. La figure 4.6 présente la barre de menu de l'interpréteur.

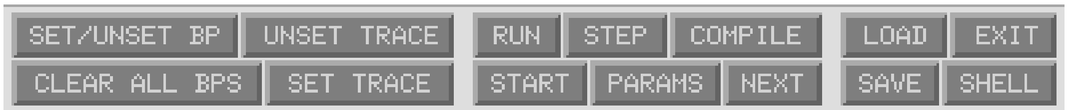


Figure 4.6: Les menus de la fenêtre d'édition-interprétation.

Les exigences de la manipulation du graphisme sont assez limités, car il n'est ici question que de visualisation — voir figure 4.7. Toutes les fonctionnalités sont accessibles à la souris.

Figure 4.7: Les menus de la fenêtre de visualisation.

La troisième et dernière fenêtre est dédiée à l'affichage des messages, qu'ils soient issus des phases de l'interprétation ou du programme. Les déplacements qui y ont lieu se font exclusivement à la souris.

4.6.2 Gestionnaire de fenêtres

GENVIEW est une application multi-fenêtres, car elle doit gérer simultanément aux moins trois fenêtres, et au plus sept. Toutes ces fenêtres sont indépendantes, et doivent éventuellement communiquer. Ceci est rendu possible par la philosophie du système **X11** qui fait que l'application n'est pas en attente des événements, mais est réveillée par leur arrivée. Cette approche est dite *event-driven* dans la littérature. Les événements sont soit des actions de l'utilisateur, déplacement de souris . . ., soit des interactions avec d'autre programmes, recouvrement d'une fenêtre par une autre par exemple, qui ont lieu de façon asynchrone. Une file d'attente est remplie par le système, et vidée par les programmes qui utilisent les fenêtres impliquées dans le déroulement d'un événement.

Pour GENVIEW la manière de procéder se fonde assez classiquement sur l'existence d'une entité virtuelle englobant toutes les fenêtres, le gestionnaire, ainsi que d'une collection de fonctions permettant le dialogue entre celui-ci et chaque fenêtre. Au cas où une fenêtre doit envoyer une information à une autre sans passer par le gestionnaire, une fonction spécifique est nécessaire. Lors de leur ouverture, les fenêtres informent le gestionnaire de leur existence. Il peut ainsi se configurer pour recevoir les événements qui les concernent, et ainsi extraire de la file pour les communiquer à la fenêtre concernée.

Chaque fenêtre gère de son côté le traitement des divers événements, car leur interprétation varie localement. Par exemple, découvrir la fenêtre de visualisation va entraîner un rafraîchissement de la zone graphique qui avait été obscurcie, alors que dans l'éditeur, la page de texte est complètement redessinée.

Le choix de principe consistant à n'utiliser que la couche **X11**[Nye88] de base pour l'implémentation de l'outil assure une rapidité d'exécution supérieure à celle obtensible à travers une bibliothèque spécialisée, comme **XView** ou **Motif**. En contrepartie, le temps de développement en est également supérieur.

4.6.3 Définition d'une structure de données rectangle pour l'affichage

La structure de données MBK est conçue pour la manipulation d'objets symboliques facilitant la conception et permettant la portabilité technologique. Elle n'est pas adaptée à la représentation graphique directe. De plus, elle n'est pas adaptée à la représentation de l'interface des cellules telle que le concepteur doit les voir pour développer en GENLIB. En effet, les objets *instances* de MBK ne contiennent ni connecteurs, ni références, ni transparences, ni boîte d'aboutement. Or lors de l'assemblage hiérarchique, ces objets sont indispensables. Afin de pallier cet inconvénient, une structure de données de plus bas niveau sémantique destinée à la représentation des rectangles est utilisée pour stocker l'image utile au concepteur des instances, et les éléments du niveau hiérarchique courant sous la forme des rectangles expansés décrits au paragraphe 2.3. En effet, afin d'avoir un temps de réponse rapide, il faut éviter d'avoir à recalculer l'expansion en rectangles des objets symboliques lors de chaque action liée à la visualisation. Ceci entraîne en contrepartie une utilisation de la mémoire plus importante.

Dans cette structure, les rectangles sont classés par niveau technologique, du plus profond, dessiné en premier, au plus élevé. À chaque niveau est associé la liste des rectangles qui y appartiennent, sous forme de liste simplement chaînée. Cette représentation simple qui nécessite le parcours de l'ensemble des objets de la figure courante pour chaque affichage est néanmoins suffisamment efficace dès que l'on utilise une approche hiérarchique[MSH90], comme celle utilisée dans les générateurs. De plus GENVIEW n'est pas un éditeur graphique mais un visualisateur. Les seules opérations nécessaires sont l'insertion d'objets, lors de l'appel aux fonctions GENLIB et la recherche des objets appartenant à la fenêtre de visualisation lors de l'affichage. La complexité de l'insertion est O(1), et celle de la vérification d'appartenance à une fenêtre O(n), ou n est le nombre de rectangles dans la structure. Il s'avère à l'utilisation que cette stratégie donne des temps de réponse satisfaisants pour l'application. Lors de l'affichage, cette structure est parcourue couche par couche, et seuls les rectangles qui intersectent la fenêtre de visualisation sont effectivement dessinés. (On cherche à n'afficher que la partie visible des rectangles, car la durée de l'affichage est directement proportionnelle à l'aire dessinée.)

Dans le processus de mise au point d'un générateur, il est utile de vérifier visuellement les interconnexions au niveau des aboutements de cellules. Pour cela, il faut pouvoir visualiser l'intérieur des cellules. Il s'agit de l'opération *peek*. Pour réaliser cette fonction, on crée une structure temporaire affichée sur l'instant, et détruite dès la fin de l'affichage.

4.7 Résultats

Sept générateurs *custom* ont été développés en deux ans à l'aide de GENVIEW. Une estimation du gain de temps apporté par GENVIEW pour la mise en œuvre des algorithmes de placement est délicate, mais les concepteurs des générateurs l'on utilisé de préférence à l'environnement standard — éditeur `vi`, compilateur **C** standard, exécution et visualisation sous éditeur graphique — également disponible.

Quelques chiffres concernant les générateurs sont exposés dans deux tableaux ci-dessous. Les entrées sont définies ainsi :

data donne les performances du circuit. Les valeurs sont issues de simulations **spice**.
vhdl represente la vue comportementale ainsi que des vecteurs de test fonctionnels.
icône est une représentation graphique de l'interface du bloc pour un éditeur de schéma.
netlist est une interface du bloc dans un format structurel, pour le routage, écrite en GENLIB.
layout est la description hiérarchique de la vue physique écrite en GENLIB.

Le tableau 4.1 présente le volume de développement, en terme de nombre de cellules et de lignes de code **C**.

Tableau 4.1: Volume de développement de quelques générateurs.

nom	nb d' options	nb de cellules	lignes de code					
			data	vhdl	netlist	icône	layout	total
banc de registres	10	81	150	2100	100	350	2600	6000
additionneur	10	53	200	90	950	350	1000	3000
décaleur à barillet	2	56	110	700	50	180	600	1900
multiplieur	5	78	200	450	—	120	1400	2600
ROM	7	100	100	200	50	110	1100	2900
RAM	5	69	—	600	—	—	1000	2500
FIFO	2	66	—	350	—	—	280	700

Les temps de développement associés sont présentés dans le tableau 4.2.

Tableau 4.2: Temps de développement de quelques générateurs.

nom	temps de développement						
	plan de masse	dessin des cellules	layout	data	vhdl	icône	netlist
banc de registres	1.5 mois	4 mois	4 mois	2 mois	14 jours	1 jour	1 jour
additionneur	1 mois	3 mois	15 jours	1 mois	2 jours	2 jours	1 jour
décaleur à barillet	1 mois	1.5 mois	1 mois	15 jours	2 jours	1 jour	1 jour
multiplieur	1.5 mois	2 mois	1.5 mois	1 mois	4 jours	1 jour	1 jour
ROM	2 mois	3 mois	1 mois	1 mois	1 jours	1 jour	1 jour
RAM	2 mois	3 mois	1.5 mois	1.5 mois	7 jours	—	—
FIFO	1 mois	2 mois	1 mois	1.5 mois	4 jours	—	—

- les valeurs pour la vue data sont très longues à obtenir, car elles sont issues de simulations **spice**.

- les temps de développement indiqués sont des *estimations* du temps nécessaire à la définition et à la mise au point des différents programmes, ramenés à un seul concepteur. Ces temps sont des déductions *a posteriori*, car dans les faits, les tâches sont exécutées en paralèlle. Par exemple, les cellules de bibliothèque évoluent pendant la conception, ce qui influe sur le code du générateur, et réciproquement. Ceci explique l'absence d'un temps total pour chacun de ces générateurs.

- les compétences en programmation des concepteurs de ces générateurs sont très variées, et en ce sens les temps et le nombre de lignes indiqués ici ne sont pas forcément représentatifs de la complexité du générateur. Cependant, il est intéressant d'avoir un échantillon représentatif d'une population de concepteurs.

- de manière générale, il faut deux à trois jours pour dessiner une cellule optimisée valide dans son contexte d'aboutement à partir de son schéma. Cependant, dans les générateurs réguliers, on est souvent amener à modifier une cellule existante pour gérer un effet de bord, et dans ce cas une ou deux heures sont suffisantes pour créer et valider une nouvelle cellule.

Les sources de GENVIEW font à peu près 22.000 lignes de code. Elles sont réparties ainsi : 3200 pour l'éditeur, 7300 pour l'interpréteur, 6600 pour la visualisation, les 4900 restantes étant dédiées à des tâches de gestion de fenêtres et de communication entre les différents modules. 800 lignes de **RTL** et 800 lignes de **C** sont nécessaire à la paramétrisation de la machine virtuelle pour **gcc**.

4.8 Conclusion

La spécification de l'outil GENVIEW est le résultat d'une expérience concrète réalisée chez BULL, où le développement et surtout la mise au point d'un générateur de *ROM* paramétrable en langage **C** avait demandé un effort supérieur à deux hommes $\times$ an.

Les points forts de GENVIEW pour les utilisateurs sont :

- intégration des toutes les fonctionnalités. Intrégrer les phases de saisie textuelle, d'interprétation et de visualisation sous un environnement structuré et cohérent permet au concepteur de mieux appréhender le problème et d'être ainsi plus productif.

- puissance des fonctions de mise au point. Grâce à la possibilité de tracer la valeur des variables mais surtout à la visualisation concurrente au déroulement de l'algorithme, les problèmes liés au placement des objets sont détectés très rapidement et surtout beaucoup mieux compris.

En conséquence, l'utilisation de cet environnement à permis de constater une réduction appréciable du temps de développement. Le temps d'itération, modification, compilation, exécution, visualisation passe de une demi-heure sous un environnement ou chaque outil est utilisé indépendamment à 2 minutes environ sous GENVIEW.

Ces qualités sont le résultat des principaux choix de l'implémentation :

- utlisation d'un compilateur existant paramétrable pour la génération de l'assembleur.

Le fait que ce compilateur soit paramétrable a l'avantage de permettre la définition d'une architecture cible simpliste, ce qui rend le processus d'interprétation aisé et rapide.

- mise en œuvre d'un mécanisme d' interprétation partielle . Cette approche permet de minimiser les temps d'exécution car les fonctions GENLIB ne sont pas interprétées par la machine virtuelle.

  Les appels aux fonctions système se font de manière atomique automatiquement, ce qui évite de dérouler le code de ces fonctions en mode pas à pas, ou d'avoir a gérer le problème explicitement.

  La cohérence entre l'exécution sous l'environnement graphique et sur le système hôte est garantit car il n'y a pas de duplication du code des fonctions GENLIB.

- développement graphique qui n'emploie que les fonctions de bas niveau de XWindows. Ce choix, contestable car entraînant un surcoût en temps de codage, assure en contrepartie une réelle portabilitée et une grande rapidité d'exécution, notamment sur des machines de petite puissance. De plus les besoins de GENVIEW étant relativement limité, l'utilisation d'une boîte à outils logicielle de plus haut niveau n'était pas d'un grand intérêt.

L'implémentation effective de cet outil d'une grande complexité logicielle démontre la justesse de ces choix.

Grâce à la généralité de l'approche — aucune information n'est spécifique au **MOS** — il est possible d'utiliser l'outil pour des générateurs utilisant d'autres technologies. Des études sont en cours sur l'architecture d'opérateurs en **AsGa** et en **CMOS** trois niveaux de métal.

Chapitre 5

Démonstrateur des outils

5.1 Introduction

Fournir un environnement dédié au développement de générateurs paramétrables n'a de sens que s'il s'avère efficace à l'usage. Nous présentons ici un générateur de *ROM* paramétrable. De manière générale il n'est pas inutile que le développeur d'un outil de **CAO** essaye d'utiliser son outil sur un exemple significatif. C'est encore plus vrai pour un outil interactif — comme GENVIEW — où l'automatisation des tests est impossible. Les ajustements requis, pour améliorer l'ergonomie ou corriger les *bugs* non bloquants, seront plus rapidement corrigés lorsque le développeur de l'outil s'y heurtera directement que s'ils émanent de requêtes d'autres utilisateurs . . .

La réalisation de la vue physique symbolique du générateur de *ROM* a pour but de prouver que l'ensemble GENLIB–GENVIEW, utilisant les concepts de la structure de données MBK, est un atout important pour le développement rapide de générateurs paramétrables.

Basé sur un schéma électrique et un découpage en cellules prédéfinis, issu du générateur de *ROM* `gerom` développé par une équipe de chercheurs de BULL dans le cadre du projet **Auriga**, le travail effectué dans le cadre de cette thèse a porté sur :

1. la compréhension du schéma électrique existant.
2. la modification de la bibliothèque de cellules existante pour l'adapter aux règles de dessin symboliques définies au chapitre 1.
3. le dessin d'un certain nombre de cellules originales pour des fonctionnalités nouvelles.
4. l'écriture du programme d'assemblage.
5. l'écriture des vues secondaires du générateur :
   - comportement.
   - vecteurs de test fonctionnels.
   - lecture du contenu de la *ROM* sous forme de spécification **vhdl**.
   - interface logique.

- interface physique.
- icône.
- performances — *data-sheet* —.

6. la validation d'un ensemble représentatif d'objets générés.

On entend donc par générateur un programme qui crée les vues utiles à un objet afin d'en permettre l'utilisation dans les différentes phases de la conception de circuits, et non pas uniquement sa vue physique destinée à être fabriquée.

Dans un premier temps, une méthode générale pour le développement des générateurs sera brièvement exposée, puis son application au générateur de *ROM* sera détaillée.

5.2 Aperçu de la méthode de développement des générateurs

L'écriture d'un générateur étant un travail compliqué, la mise en œuvre d'un tel projet exige une définition précise des fonctionnalités à fournir afin de satisfaire un grand nombre d'utilisateurs. Il y a un choix à faire entre versatilité, facilité d'utilisation et faisabilité. La méthode suivie suppose une connaissance de l'ensemble des paramètres qui vont être amenés à varier, ainsi que de la manière de réaliser électriquement un comportement. Par exemple s'il est possible d'utiliser la précharge, si l'on peut utiliser des transistors de passage, *etc*.

1. les spécifications externes doivent être figées. Les trois *types* de paramètres sur lesquels l'utilisateur du générateur peut jouer sont des paramètres :

   Fonctionnels, ce sont typiquement le nombre de *bits* d'entrées, la possibilité d'avoir une sortie en haute impédance . . .

   Électriques, ils jouent sur la puissance consommée, le longueur du chemin critique, la charge acceptable en sortie . . .

   Topologiques, ils concernent essentiellement le rapport largeur sur hauteur du bloc.

2. comme pour tout circuit, la première phase du travail concerne la définition des spécifications fonctionnelles. Elles sont contenues dans une description comportementale de haut niveau en langage **vhdl**. Les paramètres du générateur doivent être fixés avant de pouvoir écrire ce comportement, pour garantir notamment que l'interface du bloc est figée. Ce comportement devant être paramétrable, on utilise en fait un programme **C** qui prend comme arguments les paramètres pertinents pour cette vue, les paramètres fonctionnels, et crée le comportement qui leurs est associé. Afin d'en tester la validité, un jeu de vecteurs de test ainsi que les résultats attendus de la simulation sont générés.

3. un schéma d'interconnexions de transistors, de portes élémentaires et de connecteurs externes est ensuite défini. C'est une partie délicate, car il faut avoir à l'esprit l'usage générique qui va en être fait, et chercher la régularité sur l'ensemble des paramètres plutôt que l'optimisation sur un échantillon. C'est ici que l'on peut se rendre compte de l'impossibilité de faire varier telle ou telle fonctionnalité, et surtout que l'on doit

en définir les bornes. Typiquement, le nombre de *bits* ne pourra dépasser une certaine valeur à cause de la charge. Ceci peut entraîner un retour au point 2 pour la modification des spécifications.
Pour valider ce schéma paramétrable, il faudrait décrire sous forme textuelle, en GENLIB par exemple, les interconnexions définies sur le schéma, puis le simuler avec le même jeu de vecteurs que précédemment. Comme ce n'est qu'une phase intermédiaire de la conception, et qu'une vue liste d'interconnexions entre pseudo-cellules n'est pas une vue fournie à l'utilisateur final, on ne valide pas cette partie. Les problèmes potentiels de la synthèse manuelle seront détectés sur la liste d'interconnexions de transistors extraite du placement.

4. ainsi qu'il a été dit au paragraphe 3.3, l'approche utilisée pour la description topologique s'appuie sur une bibliothèque de cellules spécifique à chaque générateur. On va donc découper le schéma structurel en cellules de base, en utilisant éventuellement des hiérarchies intermédiaires composées de ces cellules pour construire le bloc final. Ici, il faut tenter de minimiser le nombre de cellules pour toute l'étendue des paramètres, car les problèmes liés au dessin et à la maintenance d'un nombre important de cellules sont complexes, surtout dans le cadre d'un générateur ou les contraintes de juxtapositions sont critiques. Afin de valider les objets atomiques utilisés par le placement, on va décrire les cellules sous forme de schéma de transistors d'une part et de dessin de masques d'autre part. La liste des interconnexions issue du schéma sera comparée à celle extraite des masques, et la cellule sera considérée valide lorsqu'il y aura identité. Les cellules sont censées être correctes du point de vue des règles de dessin symboliques hors de leur contexte d'utilisation. Cette *consistance* n'est pas toujours simple à obtenir, mais garantit que les problèmes de violation des règles de dessin futurs seront dus aux seules juxtapositions.
Dans un premier temps, seule l'interface topologique attendue par GENLIB peut être réalisée, laissant à plus tard la finalisation du cœur des cellules. Néanmoins, la vérification des règles d'aboutement et la vérification du comportement extrait ne peut se passer du contenu des cellules.

5. le programme d'assemblage proprement dit est alors développé. La complexité de l'assemblage varie avec le nombre et l'action des paramètres mis à disposition de l'utilisateur. Ces considérations ont normalement été abordées au point 4, car le gros du placement doit être envisagé dès ce moment la. Le dessin interne des cellules va certainement devoir évoluer pendant le développement de l'algorithme de placement, car aussi bien pensé que l'aient été les contraintes de juxtaposition, une cellule s'aboutant avec *n* voisines sera rarement correcte du premier coup. Certaines cellules de l'additionneur[LGT93] sont utilisées dans huit contextes d'aboutement différents. Afin de minimiser les temps de vérification et d'assurer le respect des règles de dessins, des programmes décrivant l'instanciation d'une cellule dans son contexte d'utilisation — avec au maximum une douzaine de voisins — sont écrits.

Comme on peut le constater, la méthode est itérative, et des boucles de mise au point sont nécessaires en phase de construction. Ceci est dû au fait qu'il faut valider la bibliothèque spécifique, au même titre que peut l'être une bibliothèque de cellules précaractérisée, avant de s'en servir.

La validation de la vue topologique va se faire en deux étapes, l'une sur le placement et l'autre sur la liste des équipotentielles extraite. Il faut tout d'abord remarquer que la validation exhaustive de tous les objets générables n'est pas possible en un temps raisonnable. Il est donc de la responsabilité du concepteur de définir un ensemble d'objets à valider qui représenterons *toutes* les configurations d'aboutements et *toutes* les familles de circuits générables. Sur cet échantillon, il faut faire les vérifications suivantes qui ont pour but de garantir le bon fonctionnement du générateur dans *tous* les cas.

1. passer un vérificateur des règles de dessin symboliques qui valide la topologie au niveau des transistors. Il est très important de ne pas vérifier uniquement la position des connecteurs des instances en vis à vis, car les problèmes sont généralement dus aux segments internes. Il suffit de regarder la cellule de la figure 3.4 page 56 pour s'en convaincre.

2. extraire les masques en une liste d'interconnexions entre transistors et connecteurs de la figure, puis vérifier cette vue. Trois possibilités pour cela :

   - simuler *switch-level* avec les vecteurs du point 2.
   - abstraire fonctionnellement le paquet de transistors en un ensemble d'équations comportementales, et les simuler avec les vecteurs du point 2.
   - abstraire fonctionnellement le paquet de transistors en un ensemble d'équations comportementales, et faire formellement la preuve que cet ensemble d'équations est équivalent aux spécifications **vhdl**.

Il est à noter que la moindre modification d'une cellule de la bibliothèque rend nécessaire la complète vérification des échantillons dans laquelle elle intervient.

Les vues satellites sont ensuite décrites pour rendre l'outil effectivement utilisable dans un contexte de conception classique, comme la vue interface logique ou la vue icône qui permettent de l'instancier pour un routage par exemple.

5.3 Un générateur de ROM

L'utilisation de la méthode décrite ci-dessus est illustrée pour `grog`, un générateur de *ROM*.

L'interface de l'outil, c'est-à-dire les paramètres accessibles à l'utilisateur, est dictée par l'expérience de concepteurs avertis, pondérée par la complexité pressentie de la bibliothèque de cellules et du programme d'assemblage. Le générateur doit pouvoir fournir des blocs :

- dont le nombre de *bits* varie. On désire pouvoir aller de 1 à 64 *bits* par mot.
- dont le nombre de mots varie. On désire pouvoir faire des blocs contenant de 64 à 4096 mots.
- qui possèdent éventuellement des transparences verticales tous les *bits*.
- dont le poids fort du bus d'adresse est soit d'indice 0, soit d'indice $\ln_2$(*nombre de mots*) - 1.
- dont le *bit* de sortie d'indice 0 se trouve soit à gauche, soit à droite du bloc.
- dont la sortie est soit simplement amplifiée, soit commandable pour être mise en haute impédance.

- dont le contenu est paramétré par un fichier en langage **vhdl** aux normes du laboratoire[BV94] dont la syntaxe est relativement souple.

  Les vues nécessaire à l'utilisation du bloc généré sont :

- la vue topologique.
- la vue interface topologique. Il n'est pas obligatoire d'avoir une vraie *ROM* pour faire des évaluations de plan de masse, et l'interface au sens GENLIB suffit. C'est en général la seule physique vue fournie par les fondeurs, car elle ne contient pas d'information technologique ou architecturale.
- la vue logique. C'est par abus de langage que cette vue est ainsi qualifiée car elle ne contient que l'interface logique utile aux routeurs. Il n'y a en aucun cas la possibilité de décrire des connexions entre instances lorsqu'il y a codage de cellules. Le codage revient à modifier l'intérieur d'une instance indépendamment des autres instances possédant le même modèle.
- la vue comportementale. Y sont associés, sur demande, les vecteurs de test exhaustifs de l'objet généré. C'est très simple et peu coûteux pour une *ROM*, puisqu'il s'agit simplement de la lecture de toutes les cases mémoire adressables.

5.3.1 Choix du schéma structurel

L'objectif de ces *ROM* est de fournir des temps d'accès très courts, et ce quelque soit le nombre de mots. Intrinsèquement, les *ROM* sont très régulières, comme on peut le constater dans [Ste77][WE85]. Elles sont donc une cible de choix pour la génération procédurale.

L'architecture n'est pas identique pour toutes les tailles de *ROM* car l'organisation interne des lignes de *bits* en fonction des lignes de mots est choisie pour minimiser le nombre de cellule, la taille des *ROM* de moins de 512 mots, et le temps d'accès et la consommation pour les grosses *ROM*s. Le schéma de la figure 5.1 est représentatif de l'architecture des blocs de 1024 mots. (Les blocs de 512 mots ont la même organisation, mais l'étage d'amplification final n'est pas préchargé.) La matrice est organisée en 32 lignes — 4 sont représentées sur le schéma — fois 16 colonnes, soit 512 mots, fois le nombre de *bits*. Les poids forts des adresses vont choisir une ligne parmi 32, et les poids faibles une colonne parmi 16. Les blocs de plus de 512 mots sont composés de sous-blocs de 512 mots dont l'adressage assure la mutuelle exclusion, ne rendant l'èvaluation active que sur un sous-bloc à la fois. Ceci permet après la première évaluation de n'avoir à préchager qu'un sous-bloc, celui qui a été évalulé sur le niveaux bas du précédent cycle d'horloge. Ainsi, la consommation d'un bloc de taille supérieure à 512 mots est identique à celle d'un bloc de 512 mots.

On remarque que c'est une architecture à précharge qui est adoptée, à la fois pour les lignes de *bits* et pour les décodeurs. Le bus de lecture est hiérarchisé en quatre étages de logique domino pour les *ROM*s de plus de 512 mots. La charge capacitive sur chacun de ces bus ne dépasse pas 32 drains de transistor, ce qui assure un temps d'accès aux données très bref. Les points mémoires sont codés en venant placer un contact connectant le drain d'un transistor avec la ligne de *bit* verticale.

Le plan de masse qui vient à l'esprit au regard du schéma structurel est présenté figure 5.2. Pour un nombre de mots supérieur a 512, $n \times 512$, on duplique n fois les parties

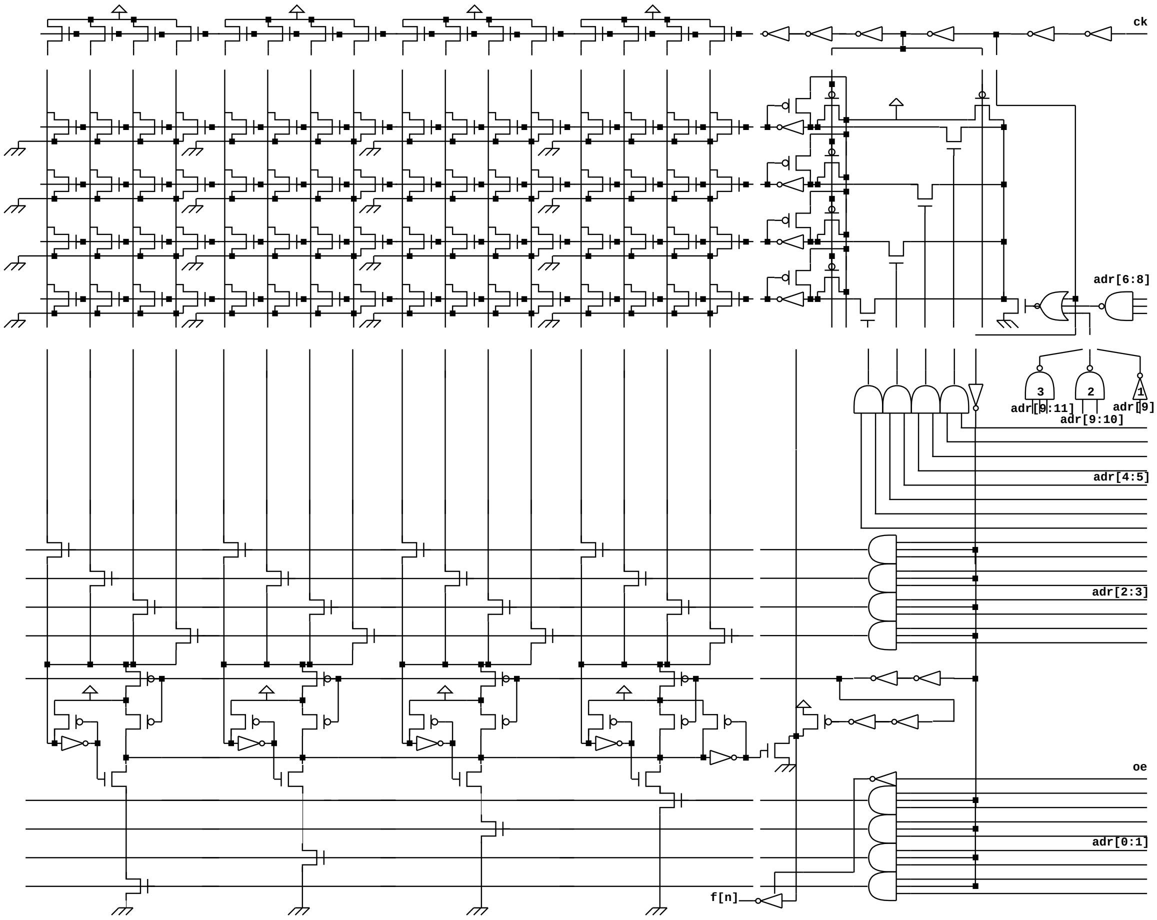


Figure 5.1: Schéma structurel des *ROM*s de 1024 mots et plus. Des schémas semblables sont disponibles en annexe G pour les quatres *familles* de la *ROM*.

précharge, matrice, polarisation, décodage des lignes de mots et horloge, en y apportant quelques modifications.

Un certain nombre de choix topologiques et électriques doivent être fait lors de la définition du plan de masse.

- on privilégie le métal1 pour les fils verticaux et le métal2 pour les fils horizontaux.
- on ne cherche pas à limiter l'utilisation des métaux pour le dessin des cellules de la bibliothèque, une option étant dédié à l'insertion de transparences. C'est différent de l'approche utilisée pour les opérateurs du chemin de données, qui doivent limiter l'utilisation du métal2 employé pour le routage au dessus des cellules.
- la polarisation du substrat des transistors codant la mémoire est faite au maximum des règles de dessin symbolique.
- les amplificateurs des lignes de mots sont prévus pour attaquer 4 fois 32 grilles de transistors, et ne sont pas variables suivant le nombre de *bits* attaqués.
- les grilles des transistors sont court-circuitées par un fil de métal. La résistance de la ligne de mots est ainsi ramenée à celle du métal.

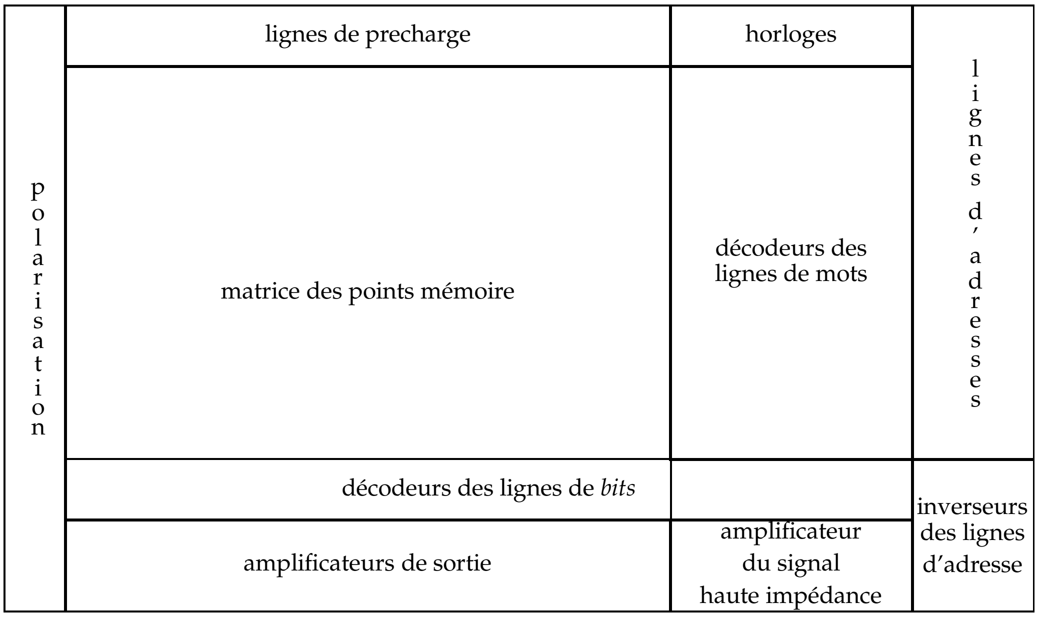


Figure 5.2: Plan de masse générique des *ROM* jusqu'à 512 mots.

- deux dimensionnements sont prévus pour les inverseurs des lignes d'adresses, le changement s'effectuant à 512 mots.

  Le comportement temporel attendu est donné figure 5.3. Le sens des délais exprimés

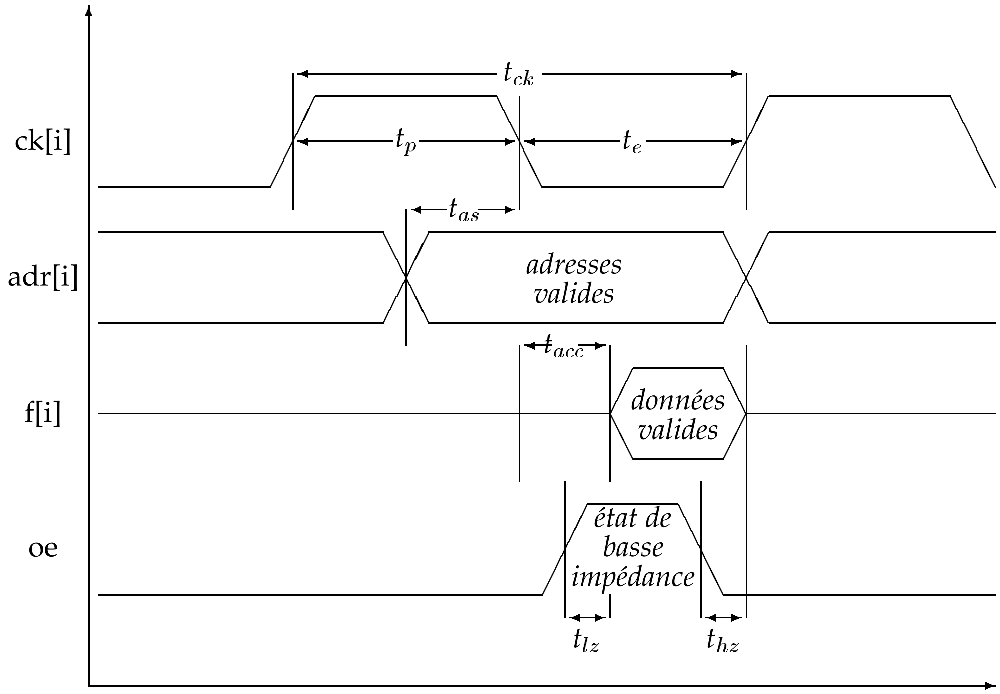


Figure 5.3: Chronogrammes des temps pertinents de la *ROM*.

sur ce chronogramme est :

Symbole	Description
t_{ck}	Temps de cycle de l'horloge
t_p	Temps de précharge, état haut de l'horloge
t_e	Temps d'évaluation, état bas de l'horloge
t_{as}	Temps de prépositionnement des adresses
t_{acc}	Temps d'accès
t_{lz}	Temps de mise en basse impédance
t_{hz}	Temps de mise en haute impédance

5.3.2 Découpage en cellules de base

À partir de tels schémas, il est possible de définir un découpage en cellules de base de manière assez visuelle. Les contraintes d'aboutement n'étant que partiellement exprimées ici, des itérations, parfois nombreuses, sont nécessaires avant d'aboutir aux cellules finales. Dans cet exemple précis, les choix du découpage sont illustrés sur la figure 5.4 qui reproduit le schéma structurel. Dans le cas de la *ROM*, les points mémoire ainsi que les étages de multiplexeurs de sorties sont répétés un grand nombre de fois, et seules ces cellules sont utilisables sur toutes l'étendue du paramètre *nombre de mots*. Trois ou quatre versions fonctionnellement identiques de la plupart des cellules sont nécessaires, car utilisées dans un contexte topologique ou électrique adapté au nombre de mots. La bibliothèque spécifique au générateur contient finalement une centaine de cellules. Il faut noter que sur ces 100

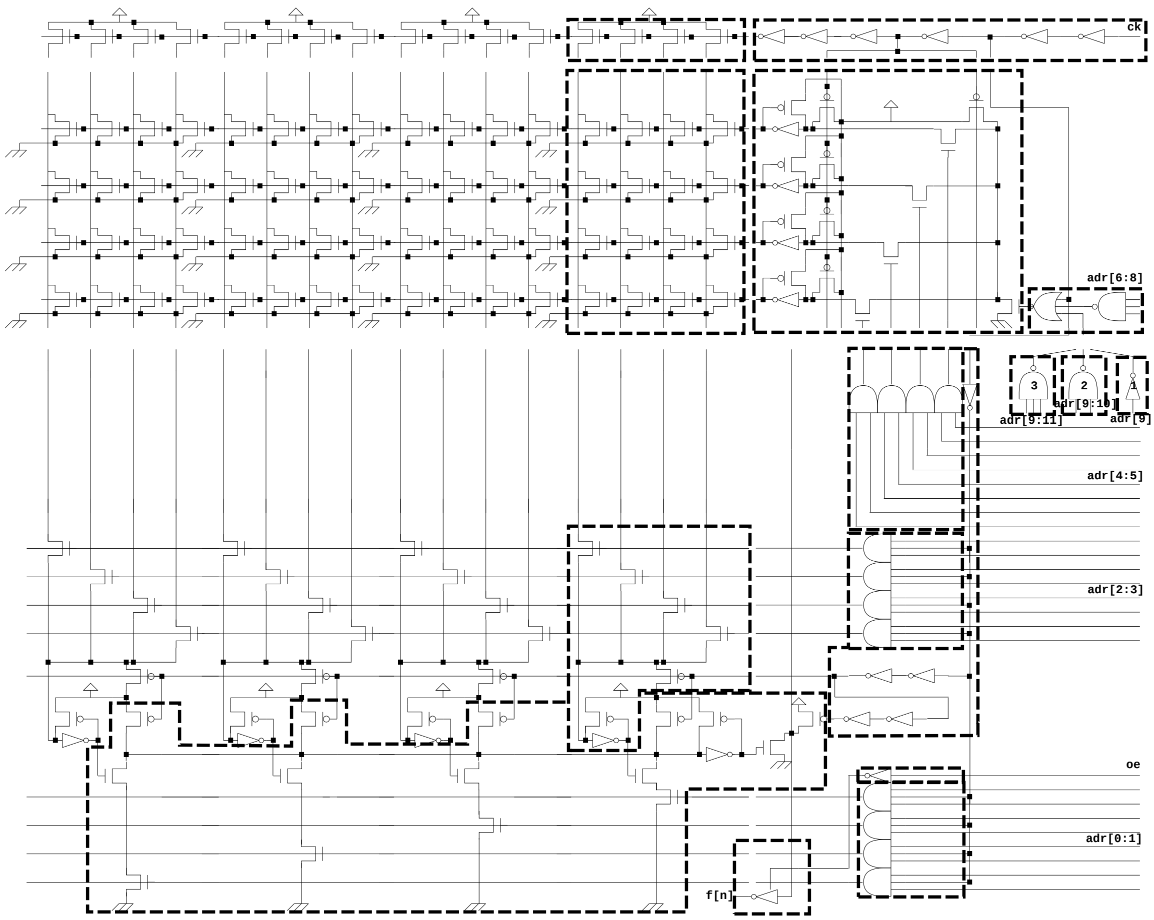


Figure 5.4: Découpage en cellules du schéma 5.1.

cellules, seules 59 contiennent des transistors, les autres servant au routage et à la polarisation de la matrice. Ces 59 cellules servent à représenter une douzaine de fonctionnalités logiques différentes. Ce sont les contraintes topologiques liées à la méthode d'assemblage qui imposent cette redondance.

Pour chacune des cellules ainsi définies, ont possède un schéma et un dessin de masques nécessaire à sa vérification fonctionnelle et topologique. Voir figure 5.5.

5.3.3 Assemblage

Si les architectures des *ROM*s sont proches en terme d'interconnexions de transistors et de topologie globale, leur assemblage est simplifié si l'on considère le générateur comme cinq sous-générateurs indépendants partageant partiellement une bibliothèque de cellules.

Les particularités de la *ROM*, du point de vue de l'écriture des générateurs sont :

- le décodage des adresses. La technique mise en œuvre repose sur l'exploitation du nom des instances chargées d'inverser les lignes d'adresses.
- le codage des valeurs dans la *ROM*. Ici, on utilise des références comme points de codage potentiels de la cellule mémoire, avec un algorithme de nommage spécifique.

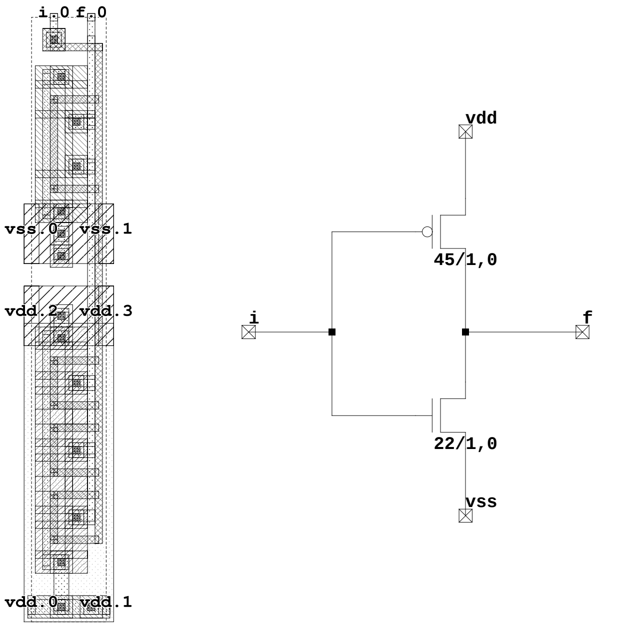


Figure 5.5: Vues utiles à la validation des cellules de base.

> Les instances de ces cellules doivent aussi être repérables aisément, afin d'avoir une fonction permettant pour un *bit* d'un mot d'adresse donnée de récupérer la référence correcte dans la bonne instance, afin de placer un contact pour court-circuiter le drain du transistor codant ce *bit* avec la ligne de *bits*.

Ces deux actions sont liées, puisque lire une valeur dans la mémoire revient à rendre actif les transistors qui auront préalablement été codés avec la valeur attendue. GENVIEW est d'une grande utilité pour vérifier la justesse des algorithmes de placement des contacts, et c'est dans ces cas délicats que l'on apprécie la souplesse d'un environnement graphique et interactif dédié à la mise au point des générateurs. On remarque sur la droite les lignes d'adresses et les contacts qui sont utilisés pour le décodage. À gauche se trouvent les points mémoires, codés également aux dessus des instances.

Un exemple plus réaliste, une *ROM* de 512 mots de 13 *bits*, figure 5.7, illustre bien les parties variant avec le nombre de *bits* et le nombre de mots lorsqu'on le compare à la figure 5.6.

5.3.4 Les vues satellites

Toutes les autres vues sont générées par des programmes en **C**.

- le comportement et les vecteurs de test sont décrits en **C**.
- l'interface logique est décrite en GENLIB.
- l'interface physique est en fait une dégradation de la vue topologique. On y a effacé tout ce qui ne fait pas partie de l'interface au sens GENLIB, définie page 56, grâce à la fonction GENLIB `OUTLINE()`.
- l'icône utilise une bibliothèque de fonctions **C** dédiées.

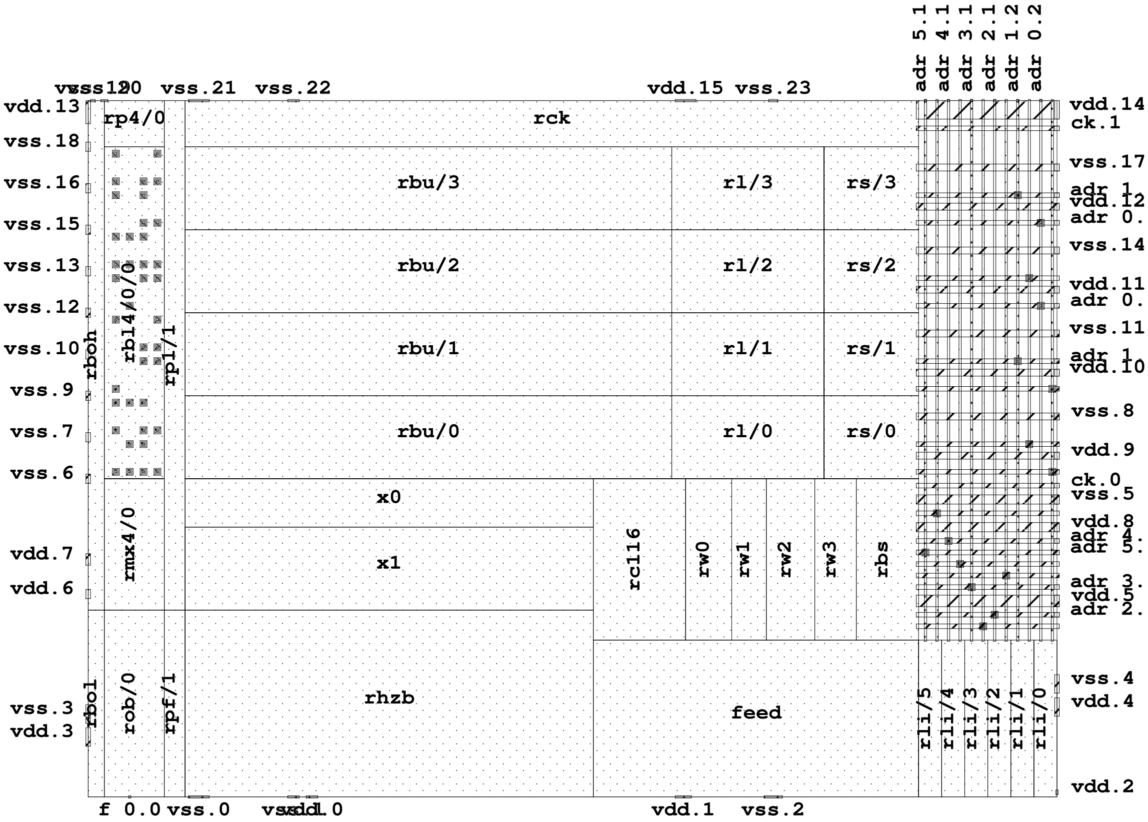


Figure 5.6: Exemple d'école, une *ROM* de 64 mots de 1 *bit*.

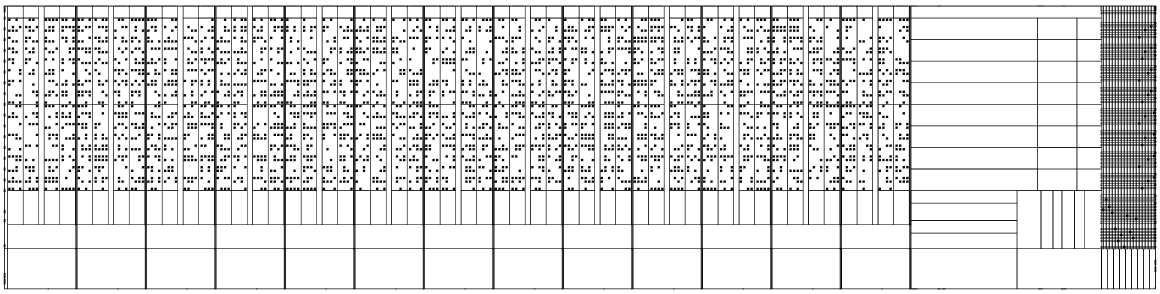


Figure 5.7: *ROM* de 512 mots de 13 *bits*.

- la vue performance correspond à la *datasheet* du bloc généré. C'est l'affichage du résultat d'une caractérisation du chemin critique sous **spice**, pour les quelques technologies accessibles au laboratoire. L'écriture de cette partie est en **C**.

5.3.5 Validation

Elle suit la méthode générale exposée paragraphe 5.2, et adopte la cosimulation pour la comparaison de la *netlist* abstraite avec le comportement. Elle se fait entièrement avec les outils de la chaîne **Alliance**. Le *Makefile* utilisé pour la validation est fourni en annexe H.

L'échantillon choisi pour valider l'ensemble des objets générables doit refléter :

1. toutes les combinaisons de juxtapositions de cellules.

2. le codage des points mémoires dans toutes les situations.
3. le décodage des adresses pour tous les nombres de mots visés.

Clairement, le point 1 est relativement simple pour la *ROM* dans le sens où pour un nombre de mots donné, le décodeur sera toujours fait de la même manière. Ceci implique la validation des onze blocs de décodeurs, correspondant aux onze tailles en mots valides — 64, 128, 256, 512, 1024, 1536, 2048, 2560, 3072, 3584 et 4096 mots — de la *ROM*.
Concernant la partie matricielles et amplification, il y a cinq architectures différentes suivant le nombre de mots, 64, 128, 256, 512 ou 1024 et plus, et suivant les paramètres *amplificateurs trois-états* et *transparences*, ce qui donne vingt possibilités. Il y a aussi des effets de bords pour des valeurs 1, 2 et 3 du nombre de *bits*, dont la croissance ne devient régulière qu'a partir de 4 *bits*. En prenant l'intersection des contraintes, on arrive à un minimum de 176 blocs — $11 \times 4 \times 4$ — à générer pour vérifier les aboutements.

Le codage, point 2, consiste à venir placer un contact sur une paire de coordonnées prédéfinies dans le dessin. Il n'est pas utile de vérifier la totalité des combinaisons — $2^{130\mathrm{k}}$ pour une taille de 4096 mots de 32 *bits* . . . —, et un remplissage plein 1 et plein 0 fait l'affaire pour le vérificateur de règles de dessin.

Le décodage des adresses est fortement lié à la manière dont sont codées les valeurs. Plusieurs méthodes sont proposées pour ces vérifications, comme écrire à chaque adresse la valeur de l'adresse, puis son inverse, et comparer à la lecture. Pour une mémoire morte, c'est une opération coûteuse, car il faut extraire le comportement de la topologie pour chaque essai. En plus de ces deux passages, un codage aléatoire sera également fait, afin de vérifier le bon comportement sur les *bits* de poids fort, car une adresse jusqu'à 4096 est codée sur 12 bits, alors que l'on désire aller jusqu'à 64 *bits*.

Finalement, le nombre minimal de blocs à vérifier pour une validation effective des objets *générables* est au minimum de $5 \times 176 = 880$. Cette procédure entièrement automatisée dure environ une semaine sur une machine **Sparc 2**.

Pour finir, il est intéressant de traduire les masques symboliques en masques réels pour le fondeur. Ceci permet de faire des tests identiques sur les masques réels que ceux qui ont été fait sur la vue physique symbolique, et de prouver par la même le bien fondé de cette approche.

Ces tests ayant été effectués avec succès, le générateur de *ROM* est validé, et peut servir comme élément constitutif d'un circuit intégré.

5.3.6 Quelques chiffres

Ce paragraphe présente des statistiques sur les objets obtenus par génération. L'évolution de divers paramètres pertinent du circuit sont affichés en fonction du nombre de mots et du nombre de *bits*, qui sont deux variables indépendantes. Ces courbes en trois dimensions sont un peu délicates à interpréter, mais elles sont révélatrices des conséquences de changements d'architectures pour un nombre de *bits* total dans la *ROM* fixe. En effet, 4 Kbits de points mémoires peuvent être organisés soit comme 128 mots de 32 *bits*, 256 mots de 16 *bits*, 512 mots de 8 *bits* ou 4096 mots de 1 *bit*. Ainsi l'étude de la courbe de la surface en fonction des 2 paramètres permet de choisir la meilleur solution. *A priori*, ce sera toujours la solution du nombre de mots minimal qui sera la meilleure, en aire, pour un nombre de *bits* total constant.

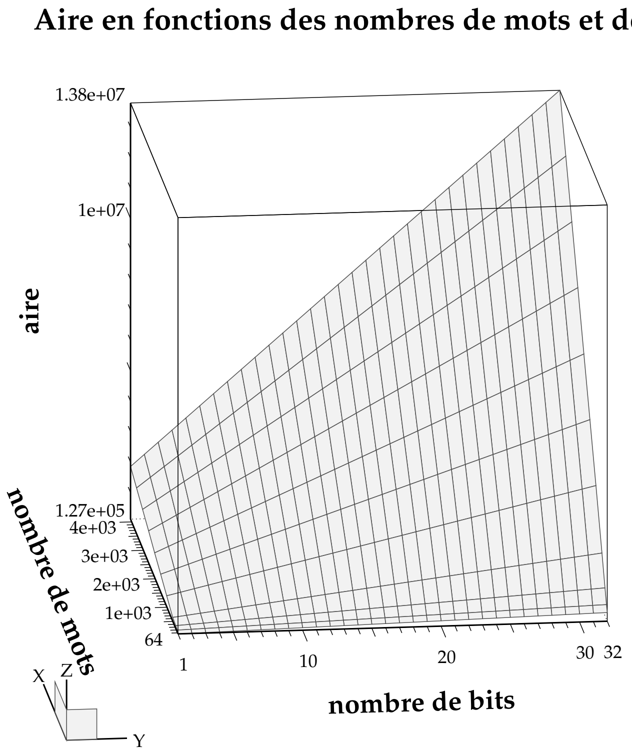


Figure 5.8: Évolution de la surface, exprimée en λ^2 en fonction du nombre de mots et du nombre de *bits* du bloc.

Tout d'abord, la figure 5.8 indique la surface en λ^2 occupée par les différents blocs générables. On constate qu'à nombre de mots constant, l'évolution en fonction du nombre de *bits* est quasiment linéaire. C'est attendu, puisqu'élargir le mot revient à ajouter une couche supplémentaire de cellules identiques pour un nombre de mots donné. Le point de départ de la courbe est représentatif de la taille du décodeur qui augmente à peu près linéairement en fonction du nombre de mots.

La figure 5.9 représente le nombre de transistors pour ces mêmes objets. Les figures 5.8 et 5.9 ont une allure similaire, ce qui n'a rien de surprenant puisque la cellule utilisée pour la mémorisation a une surface et un nombre de transistors constant.

La densité suivant les mêmes critères est affichée figure 5.10 pour une technologie cible de 1 μm. Le point mémoire est codé avec un transistor de type *N*, de taille symbolique minimale, ce qui permet d'avoir une densité d'intégration maximale. Ce qui est surtout intéressant ici est de constater que la valeur moyenne de la densité à partir d'une dizaine de *bits* est supérieure à 10^3 transistors par mm^2. Ces courbes tendent asymptotiquement vers la limite de taille imposée par la cellule mémoire, qui contient 64 transistors et à une surface de 2860 μm^2, soit une densité de 22.10^3 transistors par mm^2.

Enfin, une estimation du temps de cycle d'horloge nécessaire au bon fonctionnement de la *ROM*, toujours pour une technologie 1 μm, est présentée figure 5.11. Les fonctions ayant été utilisées pour cette évaluation sont présentées table 5.1, où *lc* est la charge en sortie :

Tableau 5.1: Formules génériques de calcul des temps de propagation d'une *ROM* pour

Nombre de transistors en fonction des nombres de mots et de bits

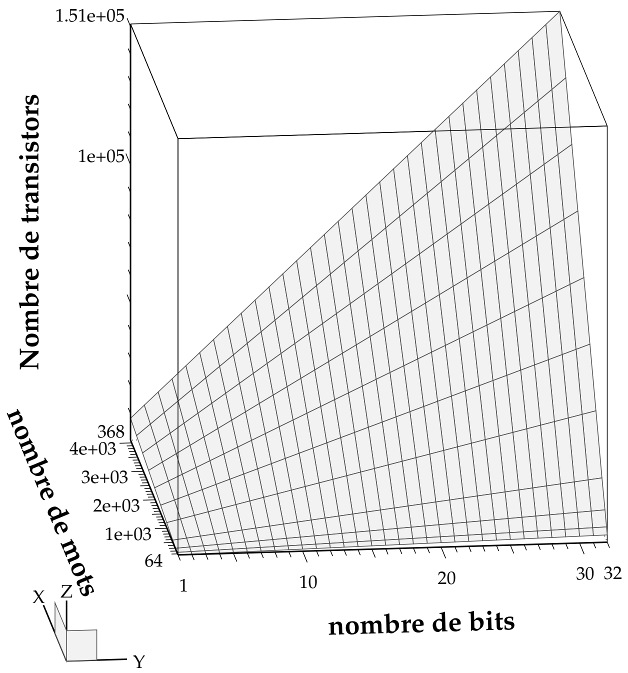


Figure 5.9: Évolution du nombre de transistors en fonction du nombre de mots et du nombre de *bits* du bloc.

une technologie donnée.

$$
\begin{aligned}
t_p &= 5.65\,10^{-9} + 4.80\,10^{-11}\,nb + 2.49\,10^{-13}\,nw \\
t_{as} &= 0.0 \\
t_{acc} &= 4.91\,10^{-9} + 8.42\,10^{-11}\,nb + 2.79\,10^{-13}\,nw + 6.01\,10^{-10}\,lc \\
t_{lz} &= 1.01\,10^{-9} + 4.46\,10^{-13}\,nb + 7.82\,10^{-14}\,nw + 1.94\,10^{-9}\,lc \\
t_{hz} &= 3.72\,10^{-9} + 1.32\,10^{-14}\,nb + 1.51\,10^{-15}\,nw + 1.06\,10^{-9}\,lc
\end{aligned}
$$

Le temps de cycle est la somme du temps de précharge et du temps d'accès.

La linéarité parfaite est ici due au modèle utilisé pour interpoler un échantillon de résultats issus de la simulation électrique fine du chemin critique. On a cherché, pour une technologie donnée, à obtenir une équation du type $t_p = t_{p_0} + \alpha_p\, nb + \beta_p\, nw$ pour tous les temps de propagations utiles. Cette approximation linéaire est justifiée par la nature de l'algorithme de construction.

Les conditions de simulations sous **spice** sont :

Charge	Alimentation	Modèle des transistors	Température
0.5 pf	4.5 V	*slow*	75^o

La totalité des sources du générateur fait 2900 lignes de code, 2300 en **C** pour la génération des vues, et 600 en **yacc** pour la lecture du sous-ensemble **VHDL** retenu pour la spécification du codage.

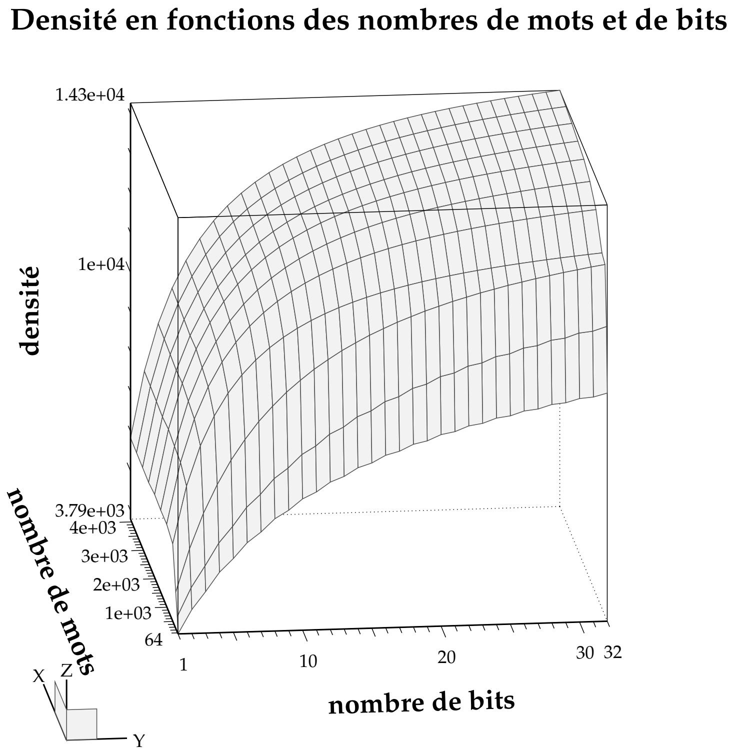


Figure 5.10: Évolution de la densité en transistor par mm^2 en fonction du nombre de mots et du nombre de *bits* du bloc pour une technologie 1 μm.

5.4 Conclusion

L'utilisation pratique du langage GENLIB et de l'environnement GENVIEW pour le développement pour ce générateur de *ROM* est utile à plus d'un titre :

- elle met en évidence la complexité algorithmique de ce type de développement. Contre toute attente, l'écriture d'un générateur de mémoire morte n'est pas triviale. Aussi régulier qu'apparaisse l'algorithme, les effets de bord, le codage des valeurs dans la matrice et la sélection des décodeurs demande une parfaite maîtrise du *layout* et des relations entre cellules.
- l'utilité du déroulement concurrent de l'algorithme de placement et de l'affichage de la figure en construction est indéniable. La mise au point du codage de la matrice a été grandement simplifiée grâce à cela.
- la description du placement n'est pas simple malgré l'utilisation du symbolique sur grille fixe qui permet de ne pas se soucier de la portabilité des contraintes d'aboutement — puisqu'elles sont garanties par la méthode —. L'utilisation d'autres méthodes, hormis le λ, engendre à coup sur un surplus important de développement logiciel.

En conséquence, cela confirme le bien fondé des choix adoptés aussi bien pour la méthode de dessin des masques que pour la spécification de l'environnement de conception dédié aux générateurs par aboutements.

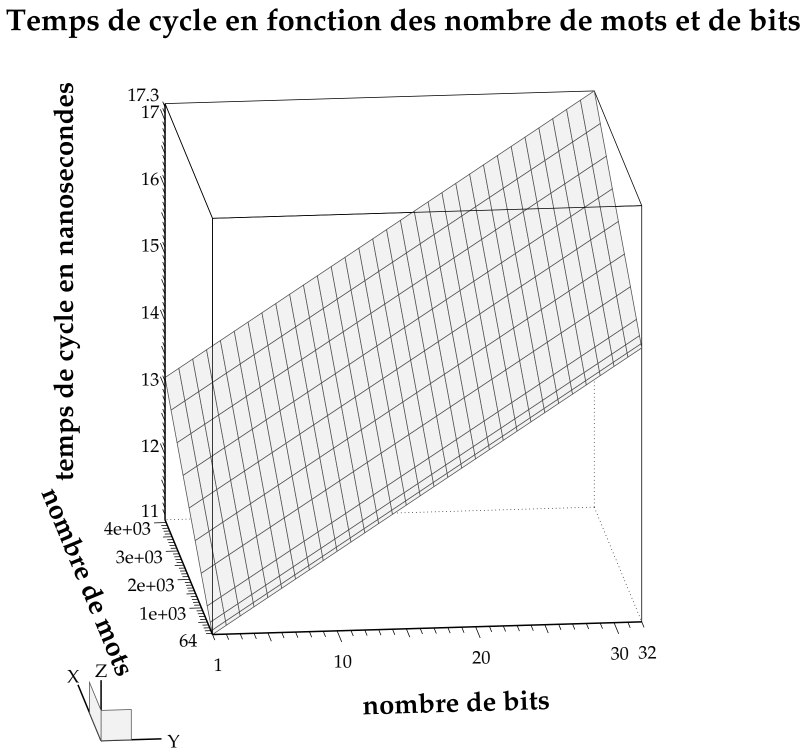


Figure 5.11: Temps de cycle en nanosecondes pour une technologie 1 μm.

Conclusion

Si l'essentiel de la recherche actuelle en micro-électronique s'intéresse aux problèmes de synthèse, elle ne doit cependant pas négliger les moyens de réaliser efficacement les implémentations sur le silicium, car l'évolution future de la compilation d'architecture dépend fortement de l'existence d'outils de synthèse logique performants et de générateurs de blocs paramétrables.

L'environnement présenté dans ce manuscrit s'est fixé comme objectif de simplifier la conception de générateurs optimisés paramétrables et portables à travers les technologies dans des temps de développement réduits. Ceci permet de fournir aux architectes et aux outils de synthèse d'architecture une riche bibliothèque de fonctions de haut niveau.

L'indépendance par rapport au processus de fabrication est garantie par une approche symbolique sur grille fixe fine, généralisant l'approche au λ. Nous avons démontré que cette approche très simple dans son principe est particulièrement bien adaptée aux circuits assemblés par aboutement. L'absence de phase de compaction garantit une conservation des alignements pour toutes les technologies cibles. L'opération de transformation de la vue symbolique en masques au micron est entièrement automatique et identique pour tous les circuits, ce qui permet l'utilisation de cette approche quelle que soit la méthodologie d'assemblage des blocs générés.

Afin d'obtenir une représentation mémoire canonique de la vue physique et de la vue structurelle des circuits intégrés, nous avons défini des structures de données qui sont à la base de la chaîne de **CAO Alliance**. L'objectif de ces structures et des fonctions qui les exploitent est de proposer une plate-forme de développement pour outils de **CAO**. Cette plate-forme doit satisfaire des critères de pédagogie et de facilité d'appréhension, ce qui est rendu possible par la définition d'une sémantique fixe des objets de la structure et une séparation claire entre la représentation de la vue physique et la représentation de la vue logique. Elle doit également assurer une relative simplicité de programmation, ce qui est fait grâce à la présence de fonctions complexes. L'ouverture sur les autres systèmes de **CAO** et le monde industriel est une nécessité. Ce but est atteint par l'acceptation de manière transparente par la structure des formats de fichiers les plus courants. Enfin, le choix d'utiliser le langage **C** pour l'implémentation nous permet de résoudre le problème de la portabilité sur diverses architectures de machines.

Le développement d'une trentaine d'outils logiciels utilisant ces structures prouve que les solutions adoptées sont pertinentes.

Le développement du langage GENLIB comme langage de description de circuits s'est appuyé sur une approche pragmatique utilisant le langage **C** enrichi de fonctions dédiées. Le but premier visé par ce langage est de permettre le développement dans des laps de temps assez court de générateurs de blocs optimisés portables. Pour cela, nous

nous sommes reposés sur l'approche symbolique sur grille fixe. Nous avons choisi de faire une nette séparation entre les cellules de base et l'assemblage hiérarchique : les cellules sont saisies sous un éditeur de masques symbolique, de la manière habituelle aux concepteurs, et seul l'assemblage se fait par programme. Les concepteurs de circuits n'étant *a priori* pas des programmeurs, un effort a été fait pour simplifier les fonctions du langage, notamment en limitant aux chaînes de caractères et aux entiers les types de données.

À l'expérience nous avons montré que ces choix sont payants, car un important nombre d'opérateurs optimisés ont été développé avec ce langage dans un temps relativement court.

La nécessité de fonctionnalités de mise au point spécifiques à la génération procédurale a abouti la spécification et l'implémentation d'un outil de mise au point novateur, graphique et interactif. De même qu'il est plus facile de dessiner une cellule sous un éditeur graphique, il est plus facile de vérifier des aboutements résultant d'un algorithme programmé pendant son déroulement. Cette constatation nous a amené à développer un outil qui intègre sous un environnement cohérent les fonctionnalités d'édition textuelle, de mise au point, et de visualisation des masques, et des temps d'interprétation courts. Pour cela, nous avons opté pour l'utilisation d'un compilateur **C** existant paramétrable — utiliser un tel outil est une nécessité pour réduire les coûts de développement —. Mais nous avons choisi un compilateur qui soit capable de générer du code assembleur sur une machine virtuelle de notre choix. Nous avons ainsi pu définir une architecture très simple qui permet d'obtenir des temps de compilation et d'interprétation rapides.

Nous avons choisi un mécanisme d' interprétation partielle qui nous garantit d'une part l'interprétation atomique des fonctions de bibliothèques, et d'autre part une cohérence entre l'exécution sous l'environnement de développement et sous forme compilée, puisque le code exécuté est le même.

Ces travaux de recherche ont été menés dans le cadre industriel du projet européen ESPRIT2 **IDPS**. Ils ont abouti à une implémentation effective de plus de 40.000 lignes de **C**.

La suite à donner à ce travail va dans deux directions. L'apparition d'un troisième niveau de métal et des contacts superposables nécessite de développer de nouvelles bibliothèques. S'il est vrai que les distances centre à centre entre les segments varient de manière relativement homogène, alors il est possible de définir un jeu de règles de dessin symbolique compatible avec notre approche du dessin de masques. Dans ce cas, les structures de données et les outils de génération procédurale sont directement utilisables. On peut également en envisager l'adaptation vers une technologie symbolique **AsGa** MESFET, qui est d'ailleurs déjà à l'étude au sein du laboratoire, ou **BiCMOS**. Les principes sur lesquels se fondent l'approche symbolique sur grille fixe et les outils présentés ne sont pas bouleversés par ces technologies qui manipulent des objets proches de ceux du **CMOS**.

La seconde est essentiellement liée à l'utilisation des opérateurs générés. Que ce soit dans un contexte de conception manuel , ou dans l'utilisation par des outils de synthèse de haut niveau, il est nécessaire de définir une interface standard pour l'appel des générateurs. Ceci permettrait d'en faire une bibliothèque de fonctions **C** utilisable dans le contexte des outils de synthèse d'architecture.

Annexe A

Translation of symbolic objects into real rectangles

Authors
Frédéric Pétrot and Franck Wajsbürt
Équipe CAO-VLSI du laboratoire MASI
Tour 55-65, 201
Université Pierre et Marie Curie
4, Place Jussieu, 75252 Paris cedex 5

A.1 Introduction

Designing ASICs with the **Alliance CAD** system ensures the access to a large number of components: standard cell libraries for usual place and route tools and dedicated datapath structures, custom and semi-custom (datapath dedicated) generators and soon macrocells.

The time investment for the design of more than 600 leaf cells is very important, and therefore the layout can simply not be dedicated to a single fabrication process. Since module generaion is the most time consuming task, a simple and efficient approach to the pitch matching problem through technological retargetting must be enforced.

The goal of this paper is to describe the method that produces a process dedicated layout that does not violate foundry rules from a validated symbolic layout. It also explains how to find the values needed to parameterize the translation tool `s2r` using the symbolic design rules and the process design rules.

We use a symbolic method based upon the statistical analysis of a group of processes. Careful examination of over twenty different processes ranging from 2 μm to 0.5 μm has shown that while minimum widths and spacings are quite variable throughout the different samples, the pitches — axis to axis distances — vary more homogeneously. From these technologies and the fact that the design is mostly done using minimum width wires, a generic set of process independent symbolic design rules is defined. These rules are expressed using a unit called the λ. For a given process, the value of the λ unit is close to the minimum length of the transistor gate.

This method can be though of as a generalization of the λ as described in [MC80]. It is nevertheless quite different:

- the design is done using structured objects called *symbols*, with strong associated semantic. It is not possible to draw a transistor with polysilicon over diffusion, for example. The symbols are either defined by a single point, like contact primitives, connectors, and references, or by two points, like segments and transistors. They are placed on the nodes of an isotropic grid with a spacing of 1 λ in both directions. The layout is drawn using a simplified set of layers and symbolic rules expressed with small integers.
- the symbolic to real translation process allows to independently adjust the spacing between the objects and the size of the objects. For example, the spacing between two wires depends upon the value of the λ, while the minimum width of each wire is defined by a technological parameter different for each process layer.
- each symbol is macro-generated using technological constraints. For example, the translation of a piece of diffusion will be very different if it is a wire, a transistor drain, or part of a contact primitive.

  Three different grids are useful to explain the translations.

- the previously seen symbolic grid, with a step of 1 λ in both directions.
- the 'scaled symbolic grid', the homothetic image of the symbolic grid, with a $n \times \mu m$ step in both directions, where $n \times \mu m$ is the value of the λ for the given process. This value is computed from the set of symbolic and micron rules.
- the physical grid step, the minimum resolution of the fabrication line. It is one order of magnitude smaller than the λ. It is a technological constraint of the fabrication process.

Symbolic layout is translated into process dependent layout with the following steps:

- shrink the size of the symbolic grid to the λ value chosen for the process to obtain the 'scaled symbolic grid'. Since the grid changes homothetically, the relative positions of the centers of the symbols are kept.
- translate the symbolic objects in a collection of rectangles.
- place the instances, as this is a hierarchy preserving process.
- post process to erase notches and merge implant areas.

If the coordinates of a rectangle are not aligned on a physical grid step, then their width and height are bloated up so to snap to the physical grid.

Since the 'center' of a rectangle is snapped to a physical grid step, the extension on each side of the 'center' must be an integer number of physical grid steps. This implies that the minimum real width of a layer cannot be an odd number of physical grid steps. If such a situation occurs, the value of the width will be rounded up to the nearest greater integer number of physical grid steps. A half physical grid step translation of the center of all the rectangles is not possible because the translation is hierarchical.

This paper is organized as follows: Section A.3 assumes that all the translation tool parameters are known and explains how the translation is done. This Section is of general interest and applies for both **CMOS** and **GaAs** layouts. Section A.4 assumes that the translation formulae are known and explains how to find the parameterization values for the translation tool. This Section is valid for **CMOS** layout only.

A.2 Notations

We will use the following notations throughout the paper. The result of the transformation of a symbolic layout object is one or several user defined rectangles. A rectangle is made of:

1. a couple of coordinates, the lower left corner of the rectangle. They are called x_r, y_r.
2. dx_r, always positive, is the size of the rectangle in the x direction.
3. dy_r, always positive, is the size of the rectangle in the y direction.
4. a physical layer.

Index r indicates a 'real' value in μm and an index s indicates a 'symbolic' value in λ. p is a pitch distance, d is an edge to edge distance, and W^i s the width of the fabrication layer i. $O^{j/i}$ indicates an overlap rule of layer j over layer i, and $D^{j\ i}$ indicate a spacing rule between layer i and layer j. The min index indicates that the minimum width rule for the considered layer is used, either for a real rectangle or a symbolic segment.

Values that belong to the technology file are in upper case letters, and variables are in lower case letters.

A.3 Transformation formulae

This defines the formulae used for the translation of each symbolic object.

A.3.1 Segment transformation

A symbolic segment is defined by:

1. a couple of coordinates, the segment starting point. They are called x_s and y_s.
2. a direction, `UP`, `DOWN`, `RIGHT`, `LEFT`, called $DIRECTION$.
3. the symbolic length, l_s, always positive.
4. the symbolic width, w_s, always positive.
5. a layer.

A symbolic segment starts on (x_s, y_s), has a length l_s, in the direction $DIRECTION$, and a width of w_s. The x_s, y_s, l_s items define the axis of the segment. The rectangle has not to be centered on the axis.

To have a simpler image of the translation process, two distances are introduced, l_r and w_r, that are respectively the size of the rectangle in the axis direction, and the size of the rectangle perpendicular to the segment axis. These values are useful because they do not depend upon the segments $DIRECTION$. One will become dx_r, the other one dy_r depending on $DIRECTION$.

Each physical rectangle comes from a symbolic segment by applying a specific formula on the segment items. Two types of rectangles are necessary for the symbolic to real translation of a segment for the **CMOS** and **GaAs** target technologies we aim at.

variable width rectangles that are called `VW` in the technology file. an `VW` rectangle is used when a real rectangle has a width that depends upon the symbolic width of the segment it comes from. This is by far the most common case.

constant width rectangles. These are needed when a rectangle has a constant size regarding the segment symbolic width. Practically, this formula is needed only to compute diffusion and active areas of transistors on each side of the transistor grid. To express rectangles on both sides of the symbolic segment, two types of constant width segments are needed: the ones to be drawn at the right hand side of the symbolic segments considered as a vector, and the ones at its left hand side. They are respectively called `RCW` and `LCW`.

Note that in both cases the coordinates of the rectangle are functions of the symbolic width.

The first formula for variable width rectangles needs three user defined parameters:

DLR

physical length extension. The length of the real rectangle in the axis direction is the symbolic length times the λ plus twice DLR.

$$l_r = l_s\lambda + 2DLR$$

DWR

physical width oversize. The real width is proportional to the symbolic width times the λ, plus an algebraic user defined constant, DWR.

$$w_r = w_s\lambda + DWR$$

This is a rewritten version of equation A.1, where $DWR = W^i_{r\ min} - W^i_{s\ min}\lambda$.

$OFFSET$

physical offset, needed to shift the center of the resulting rectangle orthogonally to the segment axis. Generaly, when rectangles are centered on the segment axis, this value is zero.

The second formula for constant width rectangles also needs three user defined parameters.

DLR

physical length extension. The length of the real rectangle in the axis direction is the symbolic length times the λ plus twice DLR.

$$l_r = l_s\lambda + 2DLR$$

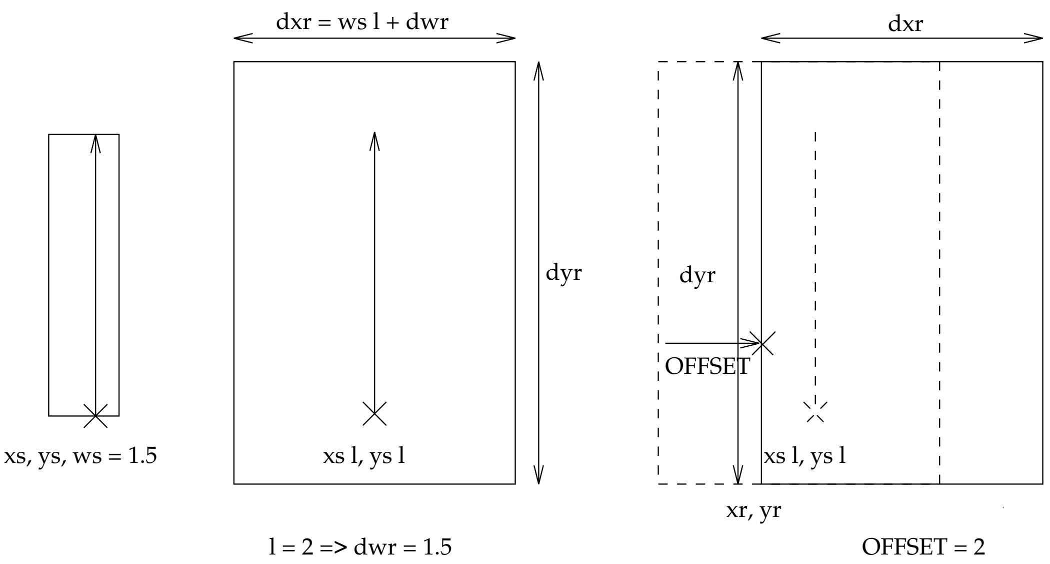
WR
: physical width. This parameters is independent of the symbolic width of the segment.

$OFFSET$
: physical offset, needed to shift the center of the resulting rectangle orthogonally to the segment axis.

The translation formulae based upon the use of these parameters are detailed in the following tables. A visual explanation of the formulae is given in the figures with each table. We are considering the symbolic to real translation of one segment with the $DIRECTION$ `UP`, and $w_s = 1.5\lambda$, the user parameters being $DWR = 1.5, DLR = 1.5, OFFSET = 2$ in all the cases. We assume the λ — noted as 'l' on the drawing — is 2 μm.

VW

$$
\begin{array}{llcl}
\texttt{UP} & x_r & = & x_s\lambda - (w_s/2)\lambda - DWR/2 + OFFSET \\
 & y_r & = & y_s\lambda - DLR \\
 & dx_r & = & w_s\lambda + DWR \\
 & dy_r & = & l_s\lambda + 2DLR \\
 & & & \\
\texttt{DOWN} & x_r & = & x_s\lambda - (w_s/2)\lambda - DWR/2 - OFFSET \\
 & y_r & = & y_s\lambda - l_s\lambda - DLR \\
 & dx_r & = & w_s\lambda + DWR \\
 & dy_r & = & l_s\lambda + 2DLR \\
 & & & \\
\texttt{RIGHT} & x_r & = & x_s\lambda - DLR \\
 & y_r & = & y_s\lambda - (w_s/2)\lambda - DWR/2 - OFFSET \\
 & dx_r & = & l_s\lambda + 2DLR \\
 & dy_r & = & w_s\lambda + DWR \\
 & & & \\
\texttt{LEFT} & x_r & = & x_s\lambda - l_s\lambda - DLR \\
 & y_r & = & y_s\lambda - (w_s/2)\lambda - DWR/2 + OFFSET \\
 & dx_r & = & l_s\lambda + 2DLR \\
 & dy_r & = & w_s\lambda + DWR
\end{array}
$$



For a symbolic to target process layout translation, it's highly probable that only this type of rectangles will be needed.

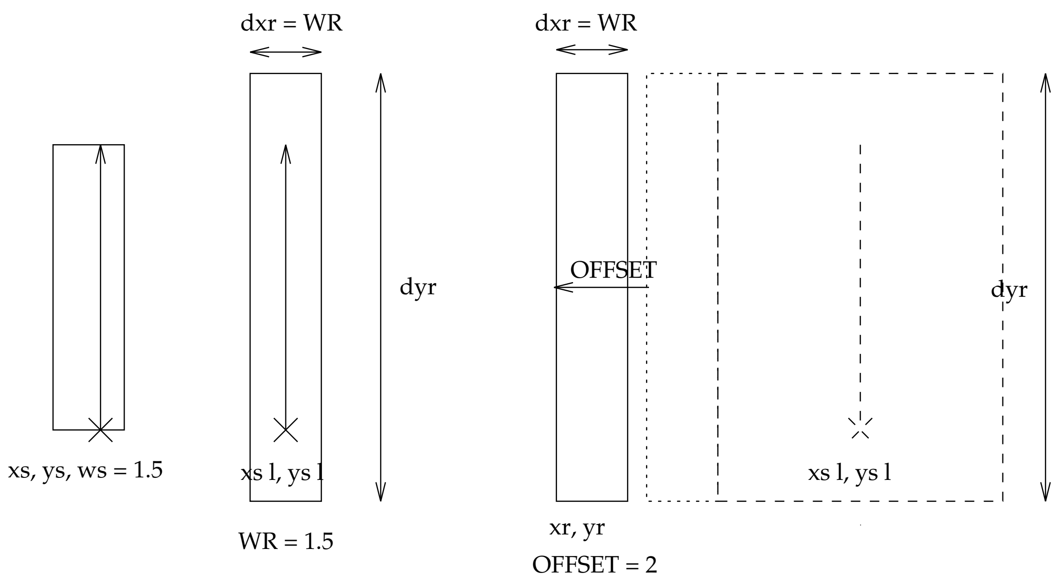
LCW

UP
$$\begin{aligned} x_r &= x_s\lambda - (w_s/2)\lambda - WR - OFFSET \\ y_r &= y_s\lambda - DLR \\ dx_r &= WR \\ dy_r &= l_s\lambda + 2DLR \end{aligned}$$

DOWN
$$\begin{aligned} x_r &= x_s\lambda + (w_s/2)\lambda + OFFSET \\ y_r &= y_s\lambda - l_s\lambda - DLR \\ dx_r &= WR \\ dy_r &= l_s\lambda + 2DLR \end{aligned}$$

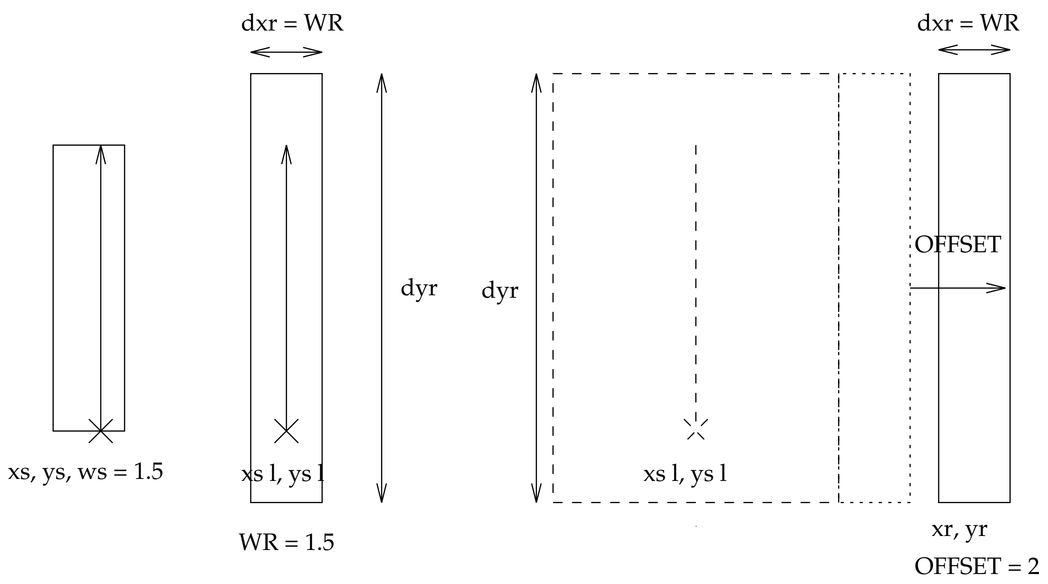
RIGHT
$$\begin{aligned} x_r &= x_s\lambda - DLR \\ y_r &= y_s\lambda + (w_s/2)\lambda + OFFSET \\ dx_r &= l_s\lambda + 2DLR \\ dy_r &= WR \end{aligned}$$

LEFT
$$\begin{aligned} x_r &= x_s\lambda - l_s\lambda - DLR \\ y_r &= y_s\lambda - (w_s/2)\lambda - WR - OFFSET \\ dx_r &= l_s\lambda + 2DLR \\ dy_r &= WR \end{aligned}$$



RCW

$$
\begin{array}{llcl}
\texttt{UP} & x_r & = & x_s\lambda + (w_s/2)\lambda + OFFSET \\
 & y_r & = & y_s\lambda - DLR \\
 & dx_r & = & WR \\
 & dy_r & = & l_s\lambda + 2DLR \\
 & & & \\
\texttt{DOWN} & x_r & = & x_s\lambda - (w_s/2)\lambda - WR - OFFSET \\
 & y_r & = & y_s\lambda - l_s\lambda - DLR \\
 & dx_r & = & WR \\
 & dy_r & = & l_s\lambda + 2DLR \\
 & & & \\
\texttt{RIGHT} & x_r & = & x_s\lambda - DLR \\
 & y_r & = & y_s\lambda - (w_s/2)\lambda - WR - OFFSET \\
 & dx_r & = & l_s\lambda + 2DLR \\
 & dy_r & = & WR \\
 & & & \\
\texttt{LEFT} & x_r & = & x_s\lambda - l_s\lambda - DLR \\
 & y_r & = & y_s\lambda + (w_s/2)\lambda + OFFSET \\
 & dx_r & = & l_s\lambda + 2DLR \\
 & dy_r & = & WR
\end{array}
$$



This type of rectangle is required for symbolic extraction purposes, as the drain and source diffusions must not be connected. It is more or less reserved to symbolic **CMOS** 'pseudo-technologies' and **GaAs** processes.

These transformations can be seen as taking place in two steps, the first one generates a rectangle with a given size, and the second one makes a translation of the lower left corner of the rectangle from a given $OFFSET$.

A.3.2 Contact transformation

A symbolic contact is defined by:

- a couple of coordinates which are the center of the contact, called x_s and y_s.
- a type.

The transformation of a contact generates generally 2 to 5 physical rectangles. A size, $SIZE$, and a layer are needed for each of these rectangles. Their coordinates are computed as follows:

$$\begin{aligned} x_r &= x_s - SIZE/2 \\ y_r &= y_s - SIZE/2 \\ dx_r &= SIZE \\ dy_r &= SIZE \end{aligned}$$

Only square contacts, or related macros such as corners for `L` and `S` shaped transistors, are allowed with this approach.

A.3.3 Connector transformation

A symbolic connector is defined by:

1. a couple of coordinates, called x_s and y_s.
2. an orientation, `NORTH`, `SOUTH`, `EAST`, `WEST`, called $ORIENTATION$.
3. a symbolic width, w_s, that is always positive.
4. a layer.

A symbolic connector produces a single rectangle. Three parameters are needed for this transformation:

- a physical layer.
- the physical width oversize, called DWR.
- the physical extension on each side of the abutment box, DER.

The formulae that apply to the connectors are, regarding the $ORIENTATION$ value:

$$\begin{array}{lrcl}
\texttt{NORTH} & x_r & = & x_s\lambda - (w_s\lambda + DWR)/2 \\
 & y_r & = & y_s\lambda - DER \\
 & dx_r & = & w_s\lambda + DWR \\
 & dy_r & = & 2DER \\
\texttt{SOUTH} & x_r & = & x_s\lambda - (w_s\lambda + DWR)/2 \\
 & y_r & = & y_s\lambda - DER \\
 & dx_r & = & w_s\lambda + DWR \\
 & dy_r & = & 2DER \\
\texttt{EAST} & x_r & = & x_s\lambda - DER \\
 & y_r & = & y_s\lambda - (w_s\lambda + DWR)/2 \\
 & dx_r & = & 2DER \\
 & dy_r & = & w_s\lambda + DWR \\
\texttt{WEST} & x_r & = & x_s\lambda - DER \\
 & y_r & = & y_s\lambda - (w_s\lambda + DWR)/2 \\
 & dx_r & = & 2DER \\
 & dy_r & = & w_s\lambda + DWR
\end{array}$$

The connector transformation formulae are derived from the segment transformation formulae since a connector is almost always placed at a segment extremity.

A.3.4 Instance transformation

A symbolic instance is defined by:

1. a couple of coordinates, the lower left corner of the instance. This point is (x_s, y_s).
2. a geometrical operation. Eight operations are possible and operation compositions give a result in the same set of operations. Operations are done 'in place', ie (x_s, y_s) indicates the lower left corner of the instanciated figure *after* the operation has been applied.
3. an instance name.
4. a model name.

The transformation of a symbolic instance into a real instance does not need any user input but the value of the λ. The coordinates in the micron world are the scaled equivalent of the symbolic ones, with a possible offset due to the different ways to represent symbolic geometrical operations, where they are done 'in place', and in the **cif** or **gdsII** format, where they are done regarding the origin, point $(0, 0)$, of the instanciated figure. This semantic change is due to the non-existence of the abutment box concept in micron layout. Let (x_1, y_1) be the lower left corner of the abutment box of the model of the instance, and (x_2, y_2) its upper right corner. x_r and y_r are the coordinates of instanciation in micron.

$$
\begin{array}{llcl}
\texttt{NOSYM} & x_r & = & (x_s - x_1)\lambda \\
 & y_r & = & (y_s - y_1)\lambda \\
\texttt{SYM\_X} & x_r & = & (x_s + x_2)\lambda \\
 & y_r & = & (y_s - y_1)\lambda \\
\texttt{SYM\_Y} & x_r & = & (x_s - x_1)\lambda \\
 & y_r & = & (y_s + y_2)\lambda \\
\texttt{SYMXY} & x_r & = & (x_s + x_2)\lambda \\
 & y_r & = & (y_s + y_2)\lambda \\
\texttt{ROT\_P} & x_r & = & (x_s + y_2)\lambda \\
 & y_r & = & (y_s - x_1)\lambda \\
\texttt{ROT\_M} & x_r & = & (x_s - y_1)\lambda \\
 & y_r & = & (y_s + x_2)\lambda \\
\texttt{SY\_RP} & x_r & = & (x_s - y_1)\lambda \\
 & y_r & = & (y_s - x_1)\lambda \\
\texttt{SY\_RM} & x_r & = & (x_s + y_2)\lambda \\
 & y_r & = & (y_s + x_2)\lambda
\end{array}
$$

A.4 How are computed the values for a given technology?

General rules can be applied to choose the right values of the formulae parameters for a given process. A program, called `prol`, is available to produce the technology file for the symbolic to real translation tool using the physical grid step, the minimum widths, spacings and overlaps of the target process. The equations used in this program are described here.

A.4.1 Computation of the λ value

The first and most important work is to determine the value of the λ, as it will define the size and overall design rule correctness of the circuit.

The λ depends on the values of the process design rules. As stated in Section A.1, the important distances in our approach are the pitches, so the value of the λ is calculated regarding centerline to centerline distances between two minimal adjacent segments. The easiest way to compute this value is to use a table with symbolic design rules and target design rules as entries and compute the ratio of 'target' over 'symbolic' for each pitch rule. The maximum value resulting from these computations gives the value of the λ. Note that all the values resulting from a symbolic to real translation are multiples of the 'physical grid' value — the resolution of the fabrication line —, and so does the λ. So each entry has to be rounded towards the nearest multiple of the physical grid.

The pitch values are for minimum width runs, as we assume that with the symbolic to micron translation formula we use, if some pitch distance rule is true for minimum width segments, then the resulting edge to edge distance rule is also true for non minimal segments.

The formula for segment transformation can be basically expressed as

$$W_r^i = (W_s^i - W_{s\ min}^i)\lambda + W_{r\ min}^i \tag{A.1}$$

For minimum symbolic width wires, it can be rewritten as $W_r = W_{r\ min}$ where it can be noticed that the value of the λ is not used.

The λ will be determined from the ratio of a real pitch distance over a symbolic pitch distance, $\lambda = p_r/p_s$. The above expression can be rewritten, using edge to edge distance and minimum width:

$$\lambda = \frac{p_r}{p_s} = \frac{d_r + W_{r\ min}^i/2 + W_{r\ min}^j/2}{d_s + W_{s\ min}^i/2 + W_{s\ min}^j/2} \tag{A.2}$$

Since this should also be true for all segments, we express the ratio of equation A.2 using the width formula for non minimal segments of equation A.1

$$\begin{aligned}\lambda &= \frac{d_r + W_r^i/2 + W_r^j/2}{d_s + W_s^i/2 + W_s^j/2} \\ &= \frac{d_r + (W_s^i - W_{s\ min}^i)\lambda/2 + W_{r\ min}^i/2 + (W_s^j - W_{s\ min}^j)\lambda/2 + W_{r\ min}^j/2}{d_s + W_s^i/2 + W_s^j/2}\end{aligned}$$

Moving λ out of the expression yields to

$$\begin{aligned}&\lambda d_s + \lambda W_s^i/2 + \lambda W_s^j/2 = \\ &\quad d_r + \lambda W_s^i/2 - \lambda W_{s\ min}^i/2 + W_{r\ min}^i/2 + \lambda W_s^j/2 - \lambda W_{s\ min}^j/2 + W_{r\ min}^j/2\end{aligned}$$

Eliminating the redundant expression on each side of the equation gives

$$\lambda d_s = d_r - \lambda W_{s\ min}^i/2 + W_{r\ min}^i/2 - \lambda W_{s\ min}^j/2 + W_{r\ min}^j/2$$

Factorizing with λ leads to

$$\lambda(d_s + W_{s\ min}^i/2 + W_{s\ min}^j/2) = d_r + W_{r\ min}^i/2 + W_{r\ min}^j/2 \tag{A.3}$$

Expression A.3 is strictly equivalent to equation A.2, thus showing that the values found for minimum geometries are applicable in the general case.

In fact this is correct only if the $W^i_{r\ min}$ and the λ are even numbers of physical grid steps. If rounding is necessary, some specific technics explained in Section A.4 solve the problem.

An example of the computation for the generic **prol12** process is given in the table below. The **edge** column displays the edge to edge symbolic design rules. The **pitch** column displays the symbolic pitch with minimum geometries. The **prol12** column contains the process edge to edge design rules. The p_r/p_s column display the pitch ratio formula. And finally, the λ column contains the resulting value rounded to the physical grid, .15 μm for this process.

rules	edge	pitch	prol12	p_r/p_s	λ
nwell	4		6		
nwell-nwell (cold)	12	12	6	$(6+6)/12=1$	1.05
dif	2		1.2		
dif-dif	3	5	2.4	$(1.2+2.4)/5=0.72$	0.75
(dif/cont)-dif	3	6	2.4	$(0.6+2.4+1.5)/6=0.75$	0.75
(dif/cont)-(dif/cont)	3	6	2.4	$(3+2.4)/6=0.9$	0.9
dif-nwell	7.5	11	3.6	$(3+3.6+0.6)/11=0.65$	0.75
nwell/dif	0.5		3.6		
difN-(difP/nwell)	9	10	7.2	$(1.2+7.2)/10=0.84$	0.9
poly	1		1.2		
Ngate	1		1.2		
Pgate	1		1.2		
poly-poly	2	3	1.8	$(1.2+1.8)/3=1$	1.05
(poly/cont)-poly	2	4	1.8	$(0.6+1.8+1.5)/4=0.97$	1.05
(poly/cont)-(poly/cont)	2	5	1.8	$(3+1.8)/5=0.96$	1.05
poly-dif	1	3	0.6	$(0.6+0.6+0.6)/3=0.6$	0.6
gate/dif	1.5		1.2		
dif/gate	2		1.8		
cont	1		1.5		
cont-cont	3	4	1.5	$(1.5+1.5)/4=0.75$	0.75
cont-gate	2	3	1.05	$(0.75+1.05+0.6)/3=0.8$	0.9
cont-dif	2	4	1.05	$(0.75+1.05+0.6)/4=0.6$	0.6
dif/cont	1		0.75		
poly/cont	1		0.75		
mt1	1		1.8		
mt1-mt1	2.5	4	1.8	$(1.8+1.8)/4=0.9$	0.9
(mt1/cont)-mt1	2.5	4	1.8	$(0.9+1.8+1.5)/4=1.05$	1.05
(mt1/cont)-(mt1/cont)	2.5	5	1.8	$(3+1.8)/5=0.96$	1.05
(mt1/via)-mt1	2.5	4	1.8	$(0.9+1.8+1.5)/4=1.05$	1.05
(mt1/via)-(mt1/via)	2.5	5	1.8	$(3+1.8)/5=0.96$	1.05
mt1/cont	0.5		0.75		
mt1/via	0.5		0.75		
via	1		1.5		
via-via	3	4	1.5	$(1.5+1.5)/4=0.75$	0.75
via-poly	2	3	1.65	$(0.75+1.65+0.6)/3=1$	1.05
via-(poly/cont)	2	4	1.65	$(0.75+1.65+1.5)/4=.975$	1.05
via-gate	2	3	1.65	$(0.75+1.65+0.6)/3=1$	1.05
via-cont	2	3	1.5	$(1.5+1.5)/3=1$	1.05
mt2	2		1.8		
mt2-mt2	2	4	1.8	$(1.8+1.8)/4=0.9$	0.9
(mt2/via)-mt2	2	5	1.8	$(0.9+1.8+1.5)/5=0.841$	0.9
(mt2/via)-(mt2/via)	2	5	1.8	$(3+1.8)/5=0.96$	1.05
mt2/via	1		0.75		

The maximum value of the last column is 1.05, so the value of the λ for this process is 1.05. Note that this value, or a very close value, should appear for the metals rules and less important the polysilicon rules, as this approach implicitly states that the size of a circuit is primary due to the metals-metals pitch rules. If it's not the case, our fixed grid approach is not well suited for the target process.

A.4.2 Contacts

The parameters for the contacts involving a cut layer are easily calculated. First compute the size of the cut, where index i is a cut layer.

$$SIZE^i = W^i_{r\ min}$$

Then, for the other layer involved in a contact, use the minimum real overlap of each layer, j, over the cut, i.

$$SIZE^j = O^{j/i}_{r\ min}$$

The $SIZE^i$ of the transistor corners — to draw `L` and `S` shaped transistors — are fixed by the rules used for a minimum width — electrical channel length — transistor.

A.4.3 Segments

A symbolic segment is a unique object, but when used for symbolic to real translation, we have to consider four types of segments:

- the runs, metals and polysilicon lines.
- the transistors and diffusions.
- the surrounding segments, the wells.

The computations must be done in that order, because each step puts constraints on the following step. We will only focus on the `VW` rectangles, as they are the only ones useful for **CMOS** processes.

Concerning the λ, two cases must be distinguished. If the λ is an even number of physical grid steps, half λ extension on each side of a segment gives an integer number of physical grid steps. If the λ is an odd number of physical grid steps, the rounding operation may create a design rule violation. When translating a non minimal segment having a width that is an odd number of λs of the most constraining rules, an edge to edge spacing error can occur because the rectangle coordinates are rounded by excess to a physical grid step, as explained Section A.1. Solving this problem is done by substracting a physical grid step to the width of all the rectangles when the λ is an odd number of physical grid steps. If the number of λs of a segment is even, the number of physical grid steps on each side of a segment is always integer, and the substracted half physical grid step on each side of the segment will be provided by the rounding function. If the λ is an odd number of physical grid steps, the substracted half physical grid step on each side of the segment will give a integer number of physical grid steps without rounding. This — always non minimal — run will have a width a physical grid step smaller than expected.

Runs

Run parameters are computed using the minimum symbolic and real width of the runs.

$$\begin{aligned} DLR^i &= \frac{1}{2} W^i_{r\ min} \\ DWR^i &= W^i_{r\ min} - W^i_{s\ min} \lambda \end{aligned}$$

DWR^i may have a negative value. In that case, two abutted colinear symbolic segments may not be connected anymore after translation. The post processing described in Section A.4.5 merges them back.

These rules are correct for the symbolic layers used in wires that have a direct equivalent in the process, namely `POLY`, `ALU1`, `ALU2`, and `ALU3`.

diffusions

DWR^i is computed as follows for the active area in runs — `NDIF`, `PDIF`, `NTIE` and `PTIE`—. The choice is to have a 3 λ run that is as large as a symbolic diffusion contact.

$$DWR^{act} = W^{con}_{r\ min} + 2\, O^{act/con}_{r\ min} - \lambda$$

DLR^{act} is based on the constraints of the transistor.

Transistors

Please note that when speaking of a transistor, the term length refers to it electrical channel length. The same applies to its width. Nevertheless, when the minimum width of a layer is required, we refer to the technological constraint.

DLR^{gat} is fixed to 0, because the computation of the λ value makes this hypothesis. One can see on layout (1) of Figure A.1 that DLR must satisfy

$$-\frac{1}{2} W^{pol}_{r\ min} \leq DLR^{gat\ in\ tr} \leq \frac{1}{2} W^{pol}_{r\ min}$$

DWR^{gat} is the same as in the runs:

$$DWR^i = W^i_{r\ min} - W^i_{s\ min} \lambda$$

This is required in order to use the minimum gate length of the process.

$DLR^{act\ in\ tr}$ has to be computed from five constraints. It is always a negative value.

- the ratio W_P/W_N must be the same than in symbolic if possible, so $W_r = n\lambda + 2DLR^{act\ in\ tr}$ must be equal to $W_s = (n-3)\lambda$.

  $$n\lambda - 2|DLR^{act\ in\ tr}| = (n-3)\lambda$$

  leads to $DLR^{act\ in\ tr} = \frac{3}{2}\lambda$.

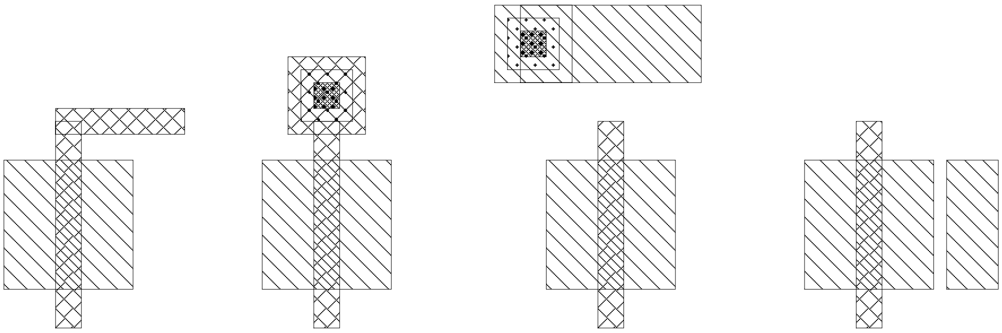
- the spacing between polysilicon and active must be respected. This spacing is illustrated in layout (1) of Figure A.1.

$$|DLR^{act\ in\ tr}| \geq \frac{1}{2}W_r^{pol} + D_{r\ min}^{pol\ act}$$

- the spacing between contact of poly and active must be respected. This spacing is shown in layout (2) of Figure A.1.

$$|DLR^{act\ in\ tr}| \geq \frac{1}{2}W_{r\ min}^{con} + O_{r\ min}^{pol/con} + D_{r\ min}^{pol\ act} - \lambda$$

Finally, $|DLR^{act\ in\ tr}|$ is the maximum of these values.



(1) (2) (3) (4)

Figure A.1: Configurations for the transistor parameters choice.

Two other spacing distances between active in a transistor and active in symbolic contacts and segments must be respected, as shown on layout (3) of Figure A.1. Since it is not possible to express these distances using $DLR^{act\ in\ tr}$, this will force the choice of DLR^{act} in wires. If we can prove the rules for the active wires, then we implicitly prove them for the active into transistors. The symbolic transistor has a length $W_s = (n-3)\lambda$, so we express DLR^{act} as

$$DLR^{act} = 2\lambda + DLR^{act\ in\ tr}$$

This ensures that an active wire with a length of $n \times \lambda$ is as long as the active area of a transistor with a width of $(n+4) \times \lambda$, as on layout (4), Figure A.1. (This behavior is the same as the one of symbolic rules.) To verify this constraint, we must check that DLR^{act} is smaller than $\frac{1}{2}W_{r\ min}^{act}$, the value that is used when computing the λ.

The overlap of the active area around the transistor gate must be bigger than its minimum,

$$DWR_{act\ in\ tr} \geq W_{r\ min}^{gat} + 2\,O_{r\ min}^{act/gat}$$

but may be as big as its symbolic value, which is 5 λ.

The implant area and well sizes can be directly calculated. The minimum overlap rule around the active area is used. This gives:

$$\begin{aligned} DLR^{j} &= DLR^{i} + O_{r\ min}^{j/i} \\ DWR^{j} &= DWR^{i}\lambda + O_{r\ min}^{j/i} \end{aligned}$$

where j is the implant or well layer, and i is the active layer, either in a transistor or in a wire.

Wells

The determination of well parameters is a little complicated. This is because it is a segment that surrounds other segments. It is based on the exhaustive analysis of the different objects it may contain. The most constraining configurations will determine the extensions.

Figures A.2 and A.3 show the possible symbolic configurations for a technology where the well has been automatically generated with the minimum technological value.

First, DLR.

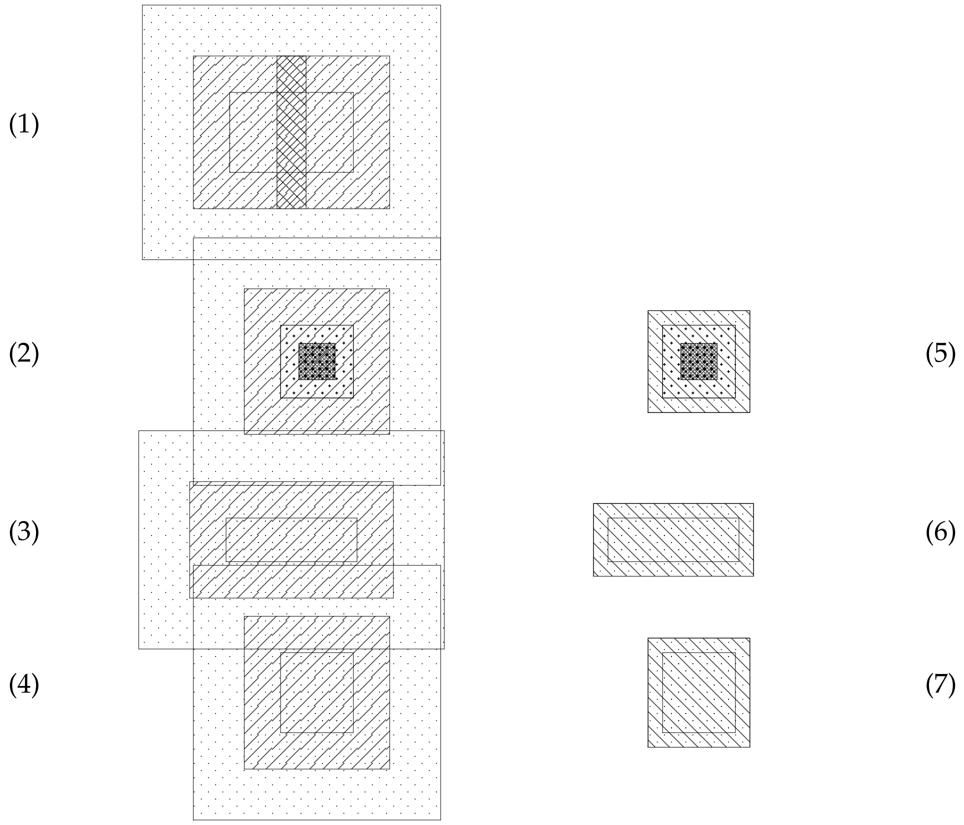


Figure A.2: Configurations for well DLR choice. All the well segments are drawn vertically.

(1) transistor is orthogonal to well:

$$O_{r\ min}^{wel/imp} + O_{r\ min}^{imp/act} - \frac{1}{2}\lambda$$

(2) contact diffusion is into well:

$$\frac{1}{2}W_{r\ min}^{con} + O_{r\ min}^{acti/con} + O_{r\ min}^{wel/imp} + O_{r\ min}^{imp/act} - 2\lambda$$

(3) diffusion run is orthogonal to well:

$$\frac{1}{2}W^{act} + O^{wel/imp}_{r\ min} + O^{imp/act}_{r\ min} - \frac{3}{2}\lambda$$

(4) diffusion run or transistor is parallel to well:

$$O^{wel/imp}_{r\ min} + O^{imp/act}_{r\ min} - DLR^{act} - \lambda$$

Configurations (5), (6) and (7) are the equivalent of configurations (2), (3) and (4), when the active is used for well polarization purposes. DLR^{wel} is given by the maximum of these values. On Figure A.2, configuration (3) is the most constraining one.

Then DWR.

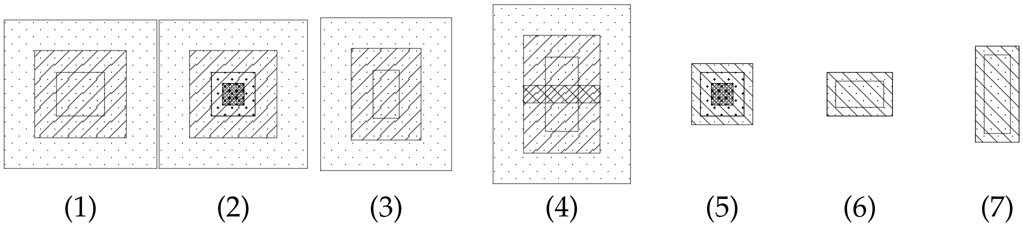


Figure A.3: Configurations for well DWR choice. All the well segments are drawn horizontally.

(1) transistor or diffusion run is orthogonal to well:

$$2DLR^{act} + 2\,O^{imp/act}_{r\ min} + 2\,O^{wel/imp}_{r\ min} - 2\lambda$$

(2) contact diffusion is into well:

$$W^{con}_{r\ min} + D^{con\ act}_{r\ min} + 2\,O^{imp/act}_{r\ min} + 2\,O^{wel/imp}_{r\ min} - 4\lambda$$

(3) diffusion run is parallel to well:

$$W^{act}_{r\ min} + 2\,O^{imp/act}_{r\ min} + 2\,O^{wel/imp}_{r\ min} - 3\lambda$$

(4) transistor parallel is to well:

$$2\,O^{imp/act}_{r\ min} + 2\,O^{wel/imp}_{r\ min} - \lambda$$

Configurations (5), (6) and (7) are the equivalent of configurations (2), (3) and (4), when the active is used for well polarization purposes. As for DLR^{wel}, DWR^{wel} is given by the maximum of these values. Configuration (4) is the most constraining on Figure A.2,

A.4.4 Connectors

The two needed parameters are computed as follows:

$$\begin{aligned} DER^i &= \frac{1}{2}W^i_{r\ min} \\ DWR^i &= W^i_{r\ min} - W^i_{s\ min}\lambda \end{aligned}$$

Note that these values are the same as the segment's ones, and that connectors are supposed to end runs only. In case connectors are required for other type of segments, the segment's DWR should be used.

A.4.5 Post processing

A straight translation using the values described above leads to an incorrect layout. Abutted colinear symbolic segments may not be connected anymore after this translation, notches have been created, and implant areas need to be merged.

A hierarchical merging phase needs to be performed in order to obtain a correct layout. In order to erase the notches, the rectangles are bloated, merged, and then shrunk. Two rectangles that are closer than the minimum spacing must be merged. Therefore the oversize value, OSV is computed as

$$OSV^i = \frac{1}{2}D^{i\ i}_{r\ min} - PGS$$

where PGS is the physical grid step.

When merging, the tool may create rectangles. As these rectangles should no be smaller than the physical design rule, the minimum width of each layer must be provided. The minimum layer width, MLW, is

$$MLW^i = W^i_{r\ min}$$

Finally, since this is a hierarchical process, some informations must be available throughout the hierarchies. Only a ring, per layer, is necessary to correct the notches on cell boundaries. The width of this ring, BRW is given by

$$BRW^i = D^{i\ i}_{r\ min}$$

Note that this BRW^i is applied with respect to the abutment box of the cell currently post processed and the physical envelops of its neighboring cells.

A.5 Conclusion

The approach described here provides a valid technological file for all the technologies we have tried. If that were not the case, or if some configuration resulting of the translation does not 'looks' fine, it is a good idea to slightly modify the values in the parameterization file, run the translation tool, and check the layout.

We mainly tune the file to have a nice rectangular layout, with as little notches as possible.

A.6 Custom cells replacement

It is possible to substitute custom cells, designed for a given process, to symbolic cells into a symbolic layout at translation time. Two needs lead to custom cell insertions: necessity to use some foundry specific objects, such as process dedicated pads, or small changes in a traduced symbolic layout to correct known DRC violations on a highly used cell that may lead to great density improvement — a *RAM* memory point in a matrix is a good example —.

1. a symbolic phantom cell must be drawn so as to abut it correctly with its neighbors, from a size and connector location point of view. This cell is the one being used for the whole symbolic design.

   Lets take the symbolic memory point of Figure A.4 as example. This cells violates

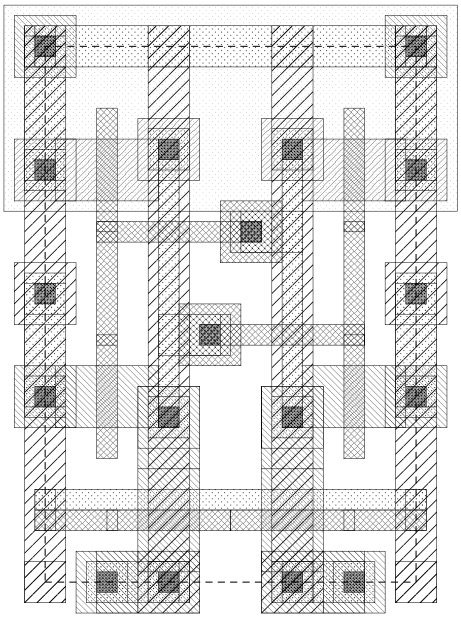


Figure A.4: Symbolic RAM memory point.

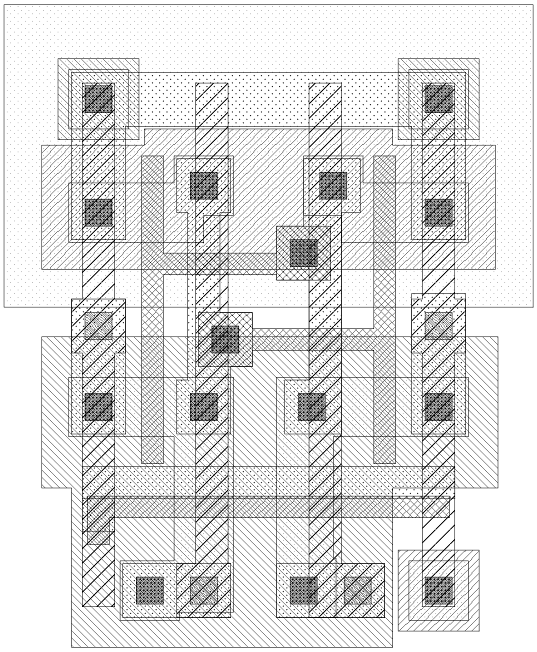


Figure A.5: Custom RAM memory point.

   two symbolic design rules: metal to metal spacing in the center of the cell. Also there is no substrate polarization inside the cell. We know that this cell must abut with itself after a symmetry around the y axis.
2. since the symbolic coordinates are snapped to a 1 λ grid, the 'real' cell will probably need to be slightly modified to ensure that its connectors will match the ones of its phantom once transformed. Computing connector locations can be done either manually with the formulae given Section A.3.3 or by running the translation tool parameterized for the process on the phantom. In any case, it is a good idea to design the symbolic cell with the lower left corner of the abutment box in $(0, 0)$ in order for the coordinates to be directly, ie without translation, usable as micron coordinate. Nevertheless, this is not required.

Since no abutment box is available in 'real', and since we use the abutment box of a model to compute the placement of its instances, the 'real' design must have its *virtual abutment box* lower left corner located in $(0, 0)$. In general, this *virtual abutment box* is the abutment box of the cell scaled by the value of the λ.

For the memory point, we wanted to put a substrate polarization contact in the lower right corner. No space was available for that, but since we know the cell abutment properties, we know we can modify the east side of the cell without interacting on its symbolic neighbors. The resulting — DRC correct — cell is shown Figure A.5.

3. add an entry in the catalog file to indicated the translation tool to substitute this cell: `phantomcell G`. The `G` stands for **gdsII** layout.
4. run the translation tool on the whole design.

Care must be taken to ensure that no notches nor that any DRC rule is violated when designing the custom cell, since no post processing will occur on either the model nor its instances.

Annexe B

Comparaison avec l'outil PRÉFORME

L'outil **PRÉFORME** utilise une grille fixe grossière qui permet en principe la conception de blocs par aboutements. Afin de comparer l'approche préconisée par **PRÉFORME** et le dessin symbolique sur grille fine sans compaction, deux cellules ont été dessinées et expansées vers une technologie cible 1 μm commune. (Pour des raisons de lisibilité, la zone d'implantation n'a pas été représentée.) Ces cellules sont issues des bibliothèques d'**Alliance**.

- la première cellule est un *nand* à deux entrées d'une bibliothèque de cellules précaractérisées. Les cellules sont placées par bandes séparées par des canaux de routages.

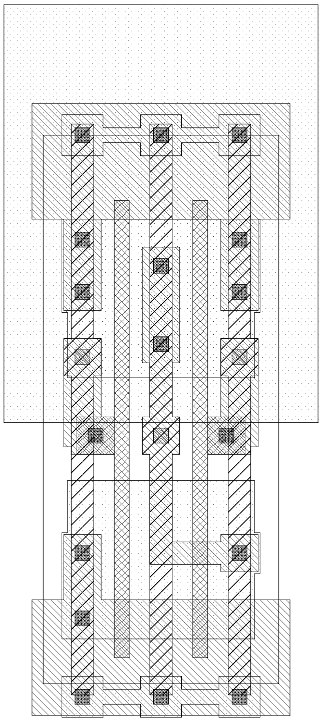


Figure B.1: *nand* en symbolique sur grille fixe expansé.

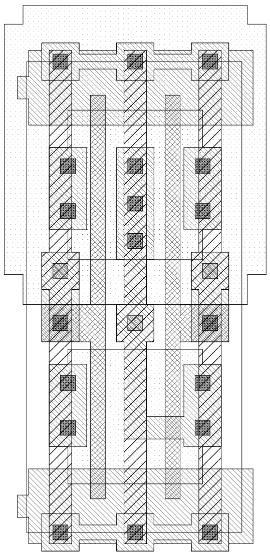


Figure B.2: *nand* dessiné sous **PRÉFORME** expansé.

La cellule sous **PRÉFORME** a une boîte d'aboutement de taille 14.375 $\mu m \times$ 31.5 μm, inférieur à celle de notre approche, 15.75 $\mu m \times$ 36.75 μm. Deux remarques à ce propos:

– **PRÉFORME** s'est autorisé à modifier la taille relative des transistors. Le rapport $\frac{W}{L}$ spécifié est le même pour les transistors P et N, or $(\frac{W}{L})_N/(\frac{W}{L})_P$ ne vaut plus 1 mais $\frac{10}{9}$. De même, le rapport $\frac{W}{L}$ vaut initialement 12, et **PRÉFORME** fournit un rapport 8. (Le résultat de notre approche donne 10.5). La porte est moins puissante que prévue. La solution consiste à ajouter un site supplémentaire en hauteur afin que l'outil puisse expanser les transistors correctement. Si l'on estime à 3.5 μm le pas entre deux sites, alors la cellule a une hauteur de 35 μm.

  L'expansion sur une technologie cible de la cellule est nécessaire s'il l'on veut se rendre compte des incohérences du dessin, car quelque soit son rapport $\frac{W}{L}$, un transistor occupe un unique site sous **PRÉFORME**.

– dans le contexte d'une bibliothèque précaractérisée, ce qui définit la hauteur de l'ensemble des cellules est un choix sur le nombre de pistes de routage interne à la cellule. Or le *nand* à deux entrées n'est pas très contraint de ce point de vue.

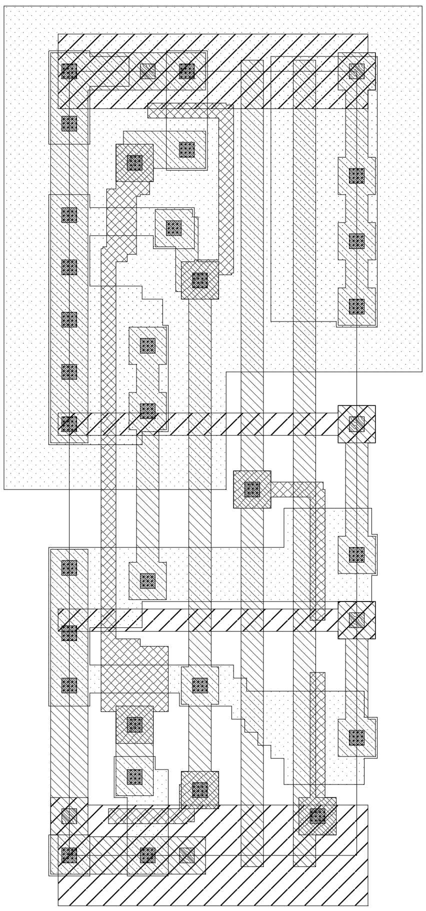


Figure B.3: Point mémoire symbolique expansé.

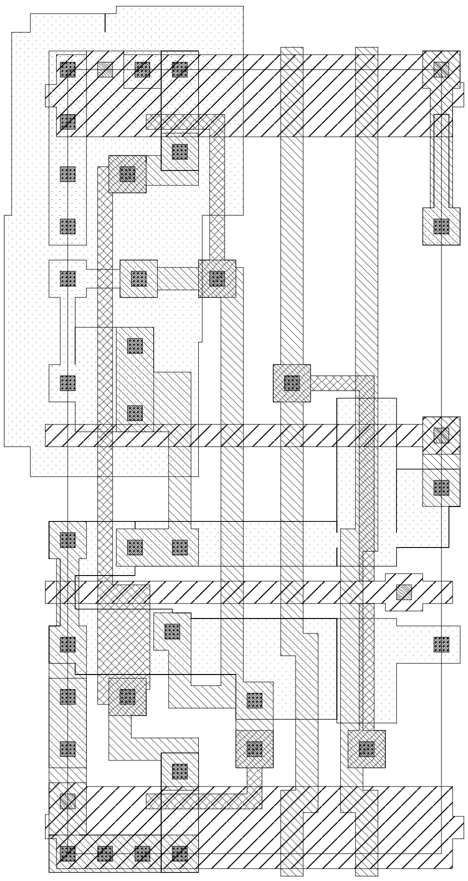


Figure B.4: Point mémoire après expansion sous **PRÉFORME**.

- la seconde cellule est le point mémoire optimisé d'un banc de registre. Elle est aboutée avec elle même dans toutes les directions, et avec d'autres cellules sur les frontières de la matrice. C'est la cellule la plus contraignante du banc de registre.

  La cellule sous **PRÉFORME** a une boîte d'aboutement de taille 25 $\mu m \times 52.5\ \mu m$, supérieure à celle de notre approche, 19.25 $\mu m \times 52.5\ \mu m$.

  Deux remarques également à ce propos :

  – le dessin des transistors de longueur de canal non minimum est pénible, car la seule solution pour obtenir un transistor affaibli est de dessiner deux transistors en parallèle avec la spécification d'un rapport $\frac{W}{L}$. Il est délicat de spécifier un rapport ainsi, et l'on obtient rarement le bon rapport après expansion. Si ce n'est pas très gênant dans le cas général, c'est particulièrement important pour les éléments mémorisants.

  – notre expérience en dessin sous **PRÉFORME** est très réduite, et il est possible qu'un *designer* habitué à l'outil obtienne une cellule plus dense. Le point de départ de la saisie est le dessin symbolique optimisé pour les règles de dessin symboliques.

La comparaison avec **PRÉFORME** s'arrête au niveau des cellules de base, car l'assemblage dans cet environnement se fait graphiquement et non procéduralement.

En conclusion, **PRÉFORME** semble être capable d'exploiter de manière assez optimale les cellules topologiquement régulières, où les fils sont droits et les transistors alignés, mais n'arrive pas à fournir d'aussi bons résultats sur de cellules sans régularité intrinsèque.

Annexe C

Structures canoniques

Cette annexe présente la description intégrale des éléments des vues physique et structurelle. Cette description est en fait l'implémentation en **C** des objets définis aux paragraphes 2.3.2 et 2.4.2.

Elle présente également des statistiques effectuées sur un échantillon représentatif de blocs qui permettent d'évaluer les besoins en mémoire de la structure en fonction du nombre d'instances ou de transistors contenus dans une figure.

C.1 Structures canoniques

Les objets composant la vue physique symbolique sont :

```
/*****************************************************************************
* layout structures types                                                    *
*****************************************************************************/
typedef struct phfig {                            /* physical figure          */
   struct phfig *NEXT;                            /* next figure              */
   struct chain *MODELCHAIN;                      /* list of models           */
   struct phseg *PHSEG;                           /* segment list head        */
   struct phvia *PHVIA;                           /* contact list head        */
   struct phref *PHREF;                           /* reference list head      */
   struct phins *PHINS;                           /* instance list head       */
   struct phcon *PHCON;                           /* connector list head      */
   char         *NAME;                            /* figure name (unique)     */
   char          MODE;                            /* if 'A' all , else perif  */
   long          XAB1,YAB1,XAB2,YAB2;             /* abutment box coordinates */
   struct ptype *USER;                            /* Application specific     */
} phfig_list;

typedef struct phcon {                            /* physical connector       */
   struct phcon *NEXT;                            /* next connector           */
   char         *NAME;                            /* connector name           */
   long          INDEX;                           /* connector index(unique)  */
   long          XCON,YCON;                       /* connector coordinates    */
   long          WIDTH;                           /* connector width          */
   char          ORIENT;                          /* possible values: N E S W */
   char          LAYER;                           /* physical layer           */
   struct ptype *USER;                            /* Application specific     */
} phcon_list;
```

```
typedef struct phseg {                       /* physical (symbolic) segment */
   struct phseg    *NEXT;                    /* next segment                */
   char            *NAME;                    /* segment name                */
   long             X1,Y1,X2,Y2;             /* segment coordinates         */
   long             WIDTH;                   /* segment width               */
   char             TYPE;                    /* possibles values: VERT HOR  */
   char             LAYER;                   /* physical layer              */
   struct ptype    *USER;                    /* Application specific        */
} phseg_list;

typedef struct phvia {                       /* physical contact            */
   struct phvia    *NEXT;                    /* next contact                */
   char             TYPE;                    /* contact type: see define    */
   long             XVIA,YVIA;               /* coordinates                 */
   struct ptype    *USER;                    /* Application specific        */
} phvia_list;

typedef struct phref {                       /* physical reference          */
   struct phref    *NEXT;                    /* next reference              */
   char            *FIGNAME;                 /* reference type              */
   char            *NAME;                    /* reference name (unique)     */
   long             XREF,YREF;               /* coordinates                 */
   struct ptype    *USER;                    /* Application specific        */
} phref_list;

typedef struct phins {                       /* physical instance           */
   struct phins    *NEXT;                    /* next instance               */
   char            *INSNAME;                 /* instance name (unique)      */
   char            *FIGNAME;                 /* model name                  */
   long             XINS,YINS;               /* coordinates                 */
   char             TRANSF;                  /* possible values: cf define  */
   struct ptype    *USER;                    /* Application specific        */
} phins_list;
```

Les éléments permettant de représenter la vue structurelle sont :

```
/*******************************************************************************
* netlist structures types                                                     *
*******************************************************************************/
typedef struct lofig {                      /* logical figure              */
   struct lofig  *NEXT;                     /* next figure                 */
   struct chain  *MODELCHAIN;               /* list of models              */
   struct locon  *LOCON;                    /* connector list head         */
   struct losig  *LOSIG;                    /* signal list head            */
   struct ptype  *BKSIG;                    /* signal block list head      */
   struct loins  *LOINS;                    /* instance list head          */
   struct lotrs  *LOTRS;                    /* transistor list head        */
   char          *NAME;                     /* figure name (unique)        */
   char           MODE;                     /* if 'A' all , else perif only */
   struct ptype  *USER;                     /* Application specific        */
} lofig_list;

typedef struct lotrs {                       /* logical transistor          */
   struct lotrs  *NEXT;                      /* next transistor             */
   struct locon  *DRAIN;                     /* drain connector             */
   struct locon  *GRID;                      /* grid connector              */
```

```
   struct locon  *SOURCE;                       /* source connector          */
   unsigned short WIDTH,LENGTH;                 /* transistor width & length   */
   unsigned short PS, PD;                       /* source & drain perimeters   */
   unsigned short XS, XD;                       /* values for area computation */
   long           X,Y;                          /* transistor coordinates      */
   char           TYPE;                         /* transistor type             */
   struct ptype  *USER;                         /* application specific        */
} lotrs_list;

typedef struct loins {                          /* logical instance            */
   struct loins  *NEXT;                         /* next figure                 */
   struct locon  *LOCON;                        /* connector list head         */
   char          *INSNAME;                      /* instance name               */
   char          *FIGNAME;                      /* model name                  */
   struct ptype  *USER;                         /* Application specific        */
} loins_list;

typedef struct locon {                          /* logical connector           */
   struct locon  *NEXT;                         /* next connector              */
   char          *NAME;                         /* connector name (unique)     */
   struct losig  *SIG;                          /* pointer on signal           */
   void          *ROOT;                         /* pointer on figure/instance  */
   char           DIRECTION;                    /* see defines for information */
   char           TYPE;                         /* instance I  / figure E      */
   struct ptype  *USER;                         /* Application specific        */
} locon_list;

typedef struct losig {                          /* logical signal              */
   struct losig  *NEXT;                         /* next signal                 */
   struct chain  *NAMECHAIN;                    /* list of alias name          */
   float          CAPA;                         /* capacitance                 */
   long           INDEX;                        /* signal index (unique)       */
   char           TYPE;                         /* internal I  / external E    */
   struct ptype  *USER;                         /* Application specific        */
} losig_list;
```

C.2 Quelques statistiques

L'occupation mémoire nécessaire à la représentation de circuits dans la structure est une quantité qu'il est intéressant de connaître *a priori*.

Afin d'avoir un sens, la mesure se base sur les hypothèses suivantes :

1. sont seuls pris en compte les blocs ayant plus de 20 instances. En effet, afin d'éviter la fragmentation mémoire, certains objets de MBK sont alloués par pages. Le chargement d'une cellule de base exige environ 44 Ko, il faut donc considérer les cas à partir desquels le coût initial est absorbé.

2. les techniques d'assemblages de blocs sont prises en compte. Les blocs construits en cellules précaractérisées n'ont pas les mêmes contraintes que les blocs construits par aboutement, et leur représentation mémoire s'en ressent.

3. les mesures portent sur des blocs soit au niveau transistors, soit au niveau hiérarchique utile pour le routage. Ce sont en effet les deux niveaux hiérarchiques exploitables, les autres n'étant introduits que pour simplifier la conception.

4. l'échantillon pour chaque type de blocs est de six blocs, contenant entre 800 et 2500 instances. Les chiffres concernant les blocs en cellules précaractérisées et chemins de données ont été obtenus sur les sous blocs du circuits **StaCS**[GLWW94]. Les blocs *customs* sont obtenus grâce aux générateurs de la chaîne **Alliance** et à un bloc *custom* de **StaCS**.

5. la mesure de la quantité de mémoire allouée dynamiquement par le processus est fournie par la fonction `mallinfo`, qui est une fonction standard d'UNIX.

Dans les tableaux, μ représente la moyenne sur l'échantillon de la quantité mesurée, et σ son écart type. Les connecteurs considérés dans la vue logique sont les connecteurs d'instances et les connecteurs de figure. Les chiffres de la vue physique s'entendent routage compris pour les blocs en cellules précaractérisées et les blocs chemin de données.

Tableau C.1: Occupation mémoire de la vue logique.

	blocs en cellules précaratérisées		blocs chemin de données		blocs *custom*
	hiérarchique	à plat	hiérarchique	à plat	à plat
μ Ko/transistor		0.189		0.175	0.175
σ		0.017		0.002	0.010
μ Ko/signal	0.391	0.377	0.407	0.428	0.431
σ	0.122	0.034	0.076	0.025	0.195
μ Ko/instance	0.429		0.486		
σ	0.130		0.097		
μ Ko/connnecteur	0.083		0.080		
σ	0.023		0.008		
μ signaux/transistor		0.500		0.411	0.475
σ		0.004		0.027	0.154

Tableau C.2: Occupation mémoire de la vue physique.

	blocs en cellules précaratérisées		blocs chemin de données		blocs *custom*
	hiérarchique	à plat	hiérarchique	à plat	à plat
μ Ko/transistor		1.658		1.561	1.222
σ		0.405		0.340	0.351
μ Ko/segment	0.155	0.134	0.128	0.128	0.131
σ	0.011	0.002	0.002	0.002	0.005
μ Ko/instance	0.726		1.953		
σ	0.236		0.365		
μ Ko/via	0.295	0.216	0.544	0.279	0.307
σ	0.044	0.019	0.091	0.019	0.047
μ segments/transistor		12.29		12.15	9.392
σ		2.758		2.720	2.905

Annexe D

Étude comparative de trois générateurs de *ROM*s

La comparaison, concernant la surface et les performances, d'objets *custom* avec des objets obtenus avec l'approche symbolique sur grille fixe n'a de sens que si l'on s'intéresse à des blocs. En effet, les contraintes imposées à une cellule dans le cadre de la génération procédurale ne permettent pas d'avoir une analyse significative sur un objet hors de son contexte d'aboutement.

On s'intéresse donc à des *ROM*s générées par trois outils : les deux premiers utilisent des règles de dessin réelles au micron, le troisième l'approche symbolique sur grille fixe suivie de l'opération de transformation aux règles réelles. Les caractéristiques physiques et temporelles des blocs issus de ces trois générateurs étant fournies pour trois technologies 1 μm différentes, l'interprétation des résultats est donc sujette à caution.

Les trois générateurs considérés sont :

`CROM01`, de la chaîne COMPASS, issu de la *1-MICRON CELL COMPILER LIBRARY* de VLSI TECHNOLOGY. Le *layout* de cette *ROM* est spécifique à la technologie 1 μm de VLSI. La programmation des valeurs dans la matrice se fait par présence ou absence de transitor.

`HARO`, de PLESSEY[PSF93]. Ce générateur est un des générateurs de *ROM* de la bibliothèque **IDPS**. La programmation des valeurs se fait en plaçant éventuellement un contact qui relie la ligne de *bit* avec un drain de transistor.

`grog`, le générateur décrit au chapitre 5 utilisant l'approche symbolique sur grille fixe. La programmation des valeurs se fait comme ci-dessus.

Une comparaison portant sur le temps développement n'est malheureusement pas possible, car aucune information de ce type n'est disponible concernant les générateurs `CROM01` et `HARO`.

Comparons tout d'abord les densités. La valeur de la densité n'est connue sur les deux générateurs dessinés au micron que pour des blocs de 64 K*bits* et plus. C'est *a priori* une valeur flatteuse, car plus le nombre de *bits* est important, moins l'impact des décodeurs des lignes de mots se fait sentir. Un calcul de pourcentages sur ces données nous informe

générateur	CROM01	HARO	grog
densité en transistors/mm^2	21000	17000	14000

que HARO est 17% plus dense que grog, et que CROM01 est elle 33% plus dense que grog. De même, CROM01 est 19% plus dense que HARO.

Mais la densité n'a pas grand sens en tant que telle. Comme toujours, l'architecture repose sur des choix entre vitesse et encombrement. Les architectures des trois générateurs sont manifestement différentes, comme le prouvent les délais ci-dessous. Examinons les temps significatifs, le temps de précharge t_p, et le temps d'accès — ou d'évaluation —, t_{acc}. Les très bonnes performances en temps d'accès aux données, t_{acc}, de grog sur un nombre

générateur	CROM01	HARO	grog
t_p (ns)	15	7	7
t_{acc} (ns)	20	15	7

de mots supérieur à 512 viennent de la hiérarchisation des bus de lecture. Il y a quatre niveaux de bus en logique domino qui minimisent le nombre de transistors sur les lignes préchargées, et donc la capacité. Lors de l'évaluation, la faible valeur de la capacité assure un temps de décharge rapide. Cette distribution a forcément un coût sur la surface, car elle nécessite des fils verticaux supplémentaires pour chaque *bits* en sortie.

En conclusion, l'utilisation du *layout* symbolique a certainement un impact sur la taille d'un bloc à architecture constante. D'après les données présentées, on peut estimer la perte en surface à maximum 20% pour une *ROM* — notons que c'est précisément ce type de blocs qui pourrait le mieux exploiter les règles au micron grâce à sa grande régularité —. En contrepartie les délais sont manifestement du même ordre de grandeur. Dans ces conditions, le prix à payer pour l'indépendance technologique est tout à fait acceptable.

Annexe E

Exemple de fichier GENLIB structurel et topologique

Ce petit exemple illustre l'utilisation de GENLIB dans le cas d'instanciation de deux blocs dont l'ordre de l'interface est connu. C'est ce fichier qui a été utilisé pour obtenir le bloc de la figure 3.8.

```
#include <genlib.h>
main()
{
   DEF_LOFIG("registerfile");
   LOCON("ad_w[0]", OUT, "ad_w[0]");
   LOCON("ad_w[1]", OUT, "ad_w[1]");
   LOCON("ad_r_b[0]", OUT, "ad_r_b[0]");
   LOCON("ad_r_b[1]", OUT, "ad_r_b[1]");
   LOCON("ck_m", OUT, "ck_m");
   LOCON("vdd", INOUT, "vdd");
   LOCON("vss", INOUT, "vss");

   LOINS("register", "register",
         "in_a[0:1]",
         "out_a[0:1]", "out_b[0:1]",
         "r_a[0:7]", "r_b[0:7]",
         "vdd", "vss",
         "w[0:7]", 0);

   LOINS("decoder", "decoder",
         "ad_r_a[0:2]", "ad_r_b[0:2]",
         "ad_w[0:2]",
         "ck_m",
         "r_a[0:7]", "r_b[0:7]",
         "vdd", "vss",
         "w[0:7]", 0);
   SAVE_LOFIG();

   DEF_PHFIG("registerfile");
   PLACE("re", "register", NOSYM, 0, 0);
   PLACE_TOP("de", "decoder", NOSYM);
   DEF_AB(0, -3, 0, 3);
   ROUTE_CHANNEL("vdd", 10, "vss", 14, 0);
   SAVE_PHFIG();
}
```

Annexe F

Aspects techniques liés au compilateur

Cette annexe présente l'ensemble du travail effectué pour la paramétrisation du compilateur **gcc** et pour l'interprétation sur la machine virtuelle. Les paragraphes F.1 et F.2 décrivent les phases nécessaire à la description d'une machine pour **gcc**. Les paragraphes F.3, F.4 et F.5 décrivent brievement l'architecture et le jeu d'instructions de la machine virtuelle.

F.1 Méthode de paramétrisation du compilateur

L'essentiel du travail du compilateur **gcc** se fait sur une représentation intermédiaire appelée **RTL**[S$^+$89], *register transfer language*, et suit les principes exposés dans [ASU86]. La description **RTL** possède deux représentations, l'une textuelle, avec une syntaxe proche du **lisp**, l'autre en mémoire, sous une forme adaptée pour le compilateur. La forme textuelle est notamment utilisée pour décrire le comportement des instructions de la machine cible du compilateur. Le langage **RTL** possède des instructions prédéfinies qui représentent les opérations atomiques comprises par le compilateur. L'utilisateur va associer à chacun de ces mots clef une ou plusieurs instructions de sa machine réelle qui vont en émuler le comportement. Dans notre cas, l'optimisation de l'allocation des registres, et des ressources en général, n'a aucun intérêt. La cible du compilateur étant une architecture virtuelle simulée, les seuls gains potentiels à l'exécution pourraient venir de simplifications du code, élimination des variables, *etc*, opérations qui ne doivent pas être effectuée lorsque l'on désire faire de la mise au point puisqu'après elles, le code intermédiaire ne reflète pas exactement le programme source. Il est à noter que la paramétrisation du compilateur n'a pas été faite dans l'optique d'obtenir un code performant, mais un code assemblable et interprétable dans un temps de développement minimum.

La première passe du compilateur consiste à lire le fichier **C** et à créer un arbre d'analyse syntaxique, tout en simplifiant les expressions arithmétiques et le calcul des constantes. Cette phase est identique quel que soit la machine cible. Ensuite, l'arbre est utilisé pour générer une représentation **RTL** correspondante. C'est ici que le compilateur va s'enquérir des instructions à sa disposition afin de choisir les stratégies idoines pour la génération de code. Pour cela, il va lire le fichier de configuration `machine.md` définissant la sémantique des instructions spécifiques à la machine en fonction des mots clef du langage **RTL**, et crée les listes **RTL** correspondantes en mémoire. Cette phase utilise deux autres fichiers qui définissent les ressources de la machine, ainsi que la manière de les utiliser. Ce sont `out-machine.c` qui fourni notamment le comportement du registre d'état lors des comparaisons, et `tm-machine.h` qui définit le nom des registres, la fonction spécifique de certain d'entre eux, comme le pointeur de pile, les prologue et épilogue des appels de fonctions, d'une certaine manière la vision externe de la machine. Le dernier fichier, `xm-machine.h`, renseigne le compilateur du compilateur sur la taille des types **C** de la machine hôte, et définit quelques fonctions dépendante de l'implémentation. La mise au point du fichier `machine.md` est délicate, et la consultation des fichiers de configuration pour d'autres architectures cibles, comme le *68000* ou le *mips*, est très utile.

Ensuite la représentation **RTL** interne est traduite en instructions assembleur puis écrite sur disque. On possède alors un fichier assembleur qui doit être traduit en binaire pour être ensuite exécuté sur la machine cible.

F.2 Exemples de définition de quelques instructions

Afin d'illustrer la méthode et les choix exposés précédemment, trois types d'opérations élémentaires sont décrites, un transfert,une opération arithmétique et un saut.

Une des instances de l'instruction `mov y x`, qui transfère l'opérande `x` dans l'opérande `y`, est examinée.

```
(define_insn "movqi"
    [(set (match_operand:QI 0 "general_operand" "=g")
          (match_operand:QI 1 "general_operand" "g"))]
    ""
    "mov.qi %0,%1")
```

La présence de `define_insn "movqi"` indique au compilateur que l'on est en train de définir l'équivalent du mot clef **RTL** `movqi`. `QI` indique que l'opération s'applique sur un octet étendu sur un entier, et correspond à un des 16 modes de représentation des types **C**, ou *machines modes*, compris par **gcc**. Le premier argument du `set` est positionné à la valeur de son second, ce qui indique clairement le transfert. La présence des `match_operand: QI` indique que les opérandes sont des quart d'entiers. L'opérande `0` doit satisfaire les conditions prévue par la fonction **C** `general_operand` définie par l'utilisateur. Dans le cas qui nous concerne, cette fonction vérifie que l'opérande est valide pour le mode en cours, `QI`, et la contrainte `g` assure que c'est soit un registre, soit une référence mémoire, soit une constante. En revanche, la contrainte = indique que l'opérande est en écriture seulement, et que son contenu est perdu après l'opération. Cela veut dire que le compilateur ne pourra présenter une constante à cette place. Ainsi, le mnémonique assembleur `mov.qi` sera utilisé pour charger dans un registre ou une case mémoire le contenu d'un registre, d'une case mémoire, ou d'une constante, selon le mode d'adressage spécifié.

Un autre exemple d'instruction, la multiplication signée double précision, décrit une opération arithmétique.

```
(define_insn "muldf3"
  [(set (match_operand:DF 0 "general_operand" "=g")
        (mult:DF (match_operand:DF 1 "general_operand" "%g")
                 (match_operand:DF 2 "general_operand" "g")))]
  ""
```

```
  "mul.df %0,%1,%2")
```

Cette instruction correspond à la multiplication de deux nombres de type `double`, indiqué par `mult: DF`, les arguments `1` et `2`, le résultat étant mis dans `0`, grâce au `set`. Comme pour le `mov.qi`, les opérandes peuvent être des constantes, des adresses, ou des registres, le résultat ne pouvant être que de l'une de ces deux dernières catégories. la contrainte `"%g"` qui est attachée à l'opérande `1` indique que l'opération est commutative, permettant ainsi d'éventuelles optimisations.

Enfin, il est intéressant de présenter une instruction de branchement conditionnel, car le registre d'état et le compteur ordinal sont concernés.

```
(define_insn "bge"
  [(set (pc)
   (if_then_else (ge (cc0) (const_int 0))
            (label_ref (match_operand 0 "" ""))
            (pc)))]
  ""
  "bge %l0")
```

Il est utile noter que toutes les instructions font progresser le compteur ordinal, `pc`, et qu'en conséquence il n'est utile de spécifier son comportement que lors des sauts. On indique ici la modification de ce registre par `set (pc)`, qui va prendre la valeur symbolique associée à l'étiquette `0`, `label_ref` dans le code ci-dessus, si le registre d'état, `cc0`, a le drapeau *plus grand ou égale à*, `ge`, qui est positionné, et garder sa valeur initiale `(pc)` dans le cas contraire. Dans l'assembleur généré, le `%l0` précise que l'opérande est une étiquette, et non une ressource de la machine.

F.3 Spécifications externes de la machine virtuelle, et conventions utilisées pour la compilation

L'architecture de la machine se veut extrêmement simple et une cible idéale du compilateur **gcc** dédié aux machines 32 *bits*. La taille naturelle des opérandes de la machine est 32 *bits*. L'ordre des octets dans un mot suit la convention qui veut que l'octet de poids fort d'un mot soit placé avant l'octet de poids faible, dite *big endian*. La machine comprend :

- 32 registres généraux de 32 *bits*.
- un compteur ordinal.
- un registre d'état.
- une pile.
- une unité arithmétique et logique entière.
- une unité de calcul flottant.
- une mémoire adressable sur 32 *bits*.

Les modes d'adressage supportés sont :

immédiat
: l'opérande attendu est une constante, comme `bra #32` qui saute à l'adresse `32`.

direct
qui considère le contenu d'une adresse, par exemple `mov.si 320,319`, qui copie le contenu de l'adresse `319` dans l'adresse `320`.

registre
qui utilise les registres internes de la machine, `add.si d31,d31,d25`, qui met dans le registre `d31` le résultat de l'addition des registres `d31` et `d25`.

indirect registre
qui utilise l'adresse pointée par le contenu d'un registre, `asl.si d4,(d3),#2`, qui met dans `d4` le résultat du décalage arithmétique par `2` de la valeur contenue dans l'adresse pointée par `d3`.

indirect registre avec déplacement
qui à le même comportement que l'indirect registre, mais avec la possibilité d'indexer l'adresse contenue dans le registre. Ceci est fort utile pour accéder à la pile des paramètres lors des appels de fonctions, comme dans `cmp.si 8(d30),#0`, qui compare à zéro la valeur du premier paramètre d'appel d'une fonction.

La taille des types **C** suit celle qui est naturelle aux machines 32 *bits* modernes, et que **gcc** préconise. Le type `char` est codé sur 8 *bits*, `short` sur 16, `integer`, `long`, `float` et `pointer` sur 32 *bits*. Les `doubles` sont étendus à 64 *bits*, et ne peuvent donc être mis dans les registres. Les types `char`, `short`, `integer`, et `long` peuvent être signés ou non signés. De plus, les `int` et les `long` sont représentés de manière strictement identique dans la machine, et dans la suite, on ne parlera plus que d'`int`. Ces types sont représentatifs des choix supportés en standard par le compilateur, et les opérations arithmétiques et logiques existent pour chacun d'eux.

Pour la compilation, on adopte le registre `d31` comme pointeur de pile, et le registre `d30` comme pointeur de contexte. Il n'y a pas d'instructions dédiées à la manipulation de la pile, et le compilateur utilise les opérateurs d'addition et de soustraction pour s'y mouvoir. La pile décroît, c'est à dire que placer une valeur dans la pile décrémente la valeur du pointeur de pile. Le pointeur de contexte, utilisé pour l'accès aux variables locales, se positionne dans la pile et croît dans le même sens que celle-ci. Cela est nécessaire pour l'implémentation de la récursion. La valeur de retour des fonctions est positionnée dans le registre `d25`.

F.4 Jeu d'instructions de la machine virtuelle

Les instructions de la machine virtuelle sont les instructions d'une machine minimale sans lesquelles le *gnu cc* ne peut être configuré. D'un point de vue temporel, les instructions sont considérées comme exécutées séquentiellement, en un cycle chacune. Tous les modes d'adressages sont supportés par ces instructions, sauf s'il est dit autrement, et travaillent sur deux opérandes, le résultat étant mis dans le troisième. Les opérations ne peuvent avoir lieu que sur des opérandes de même type, il faut donc que le compilateur utilise des conversions explicites lorsque cela est nécessaire.

Instructions logiques
Ils travaillent sur les type **C** `char`, `short` et `int` uniquement.

nom	**signification**
`and`	`et` logique
`or`	`ou` logique
`not`	`non` logique, agissant sur un opérande unique
`xor`	`ou exclusif` logique
`asl`	décalage arithmétique à gauche
`asr`	décalage arithmétique à droite
`lsl`	décalage logique à gauche
`lsr`	décalage logique à droite

Instructions de comparaison

Il y a un unique opérateur de comparaison, dont le résultat est mis dans le registre d'état. C'est d'ailleurs le seul opérateur qui puisse en modifier le contenu.

nom	**signification**
`cmp`	comparaison entre deux opérandes

Instruction de chargement

Cette instruction doit être définie pour tous les types, car c'est le seul moyen que possède le compilateur pour recopier des données.

nom	**signification**
`mov`	recopie une donnée d'une place à une autre

Instruction nulle

Il n'est jamais utilisé dans notre machine, mais doit être présent pour **gcc**.

nom	**signification**
`nop`	attend un cycle sans rien faire

Branchements

Ces instructions ne prennent qu'un opérande, qui est l'adresse à laquelle il faut sauter. Les branchements conditionnels se basent sur le registre d'état qui aura été au préalablement positionné. On ne peut faire un branchement que dans l'instruction suivant une comparaison. Pour expliquer les sauts, il faudrait dire *branchement si le drapeau du registre d'état correspondant à l'opération* `op` *est positionné* pour chacun d'eux, mais dans un but de concision, seule l'opération est spécifiée dans le tableau ci-dessous.

nom	**signification**
`beq`	égalité
`bne`	différence
`bge`	plus grand ou égal à
`bgeu`	idem, mais non signée
`blt`	plus petit que
`bltu`	idem, mais non signée
`bgt`	plus grand que
`bgtu`	idem, mais non signée
`ble`	plus petit ou égale à
`bleu`	idem, mais non signée
`bra`	branchement non conditionnel
`call`	appel de fonction. Seul l'empilement du compteur ordinal à lieu. Il est de la responsabilité de la fonction appellante de sauvegarder son contexte
`ret`	retour de fonction. Il n'y pas d'opérande, le retour prenant son argument dans la pile

Instructions de conversions

Ils permettent de passer d'un type à un autre suivant les règles définies par la norme du **C**.

nom	**signification**
`ext`	recopie d'un opérande d'un type dans un autre avec extension de signe
`ext0`	recopie d'un opérande d'un type dans un autre avec extension de zéro
`fix_trunc`	convertit un flottant en un entier signé en tronquant le résultat
`fixuns`	convertit un flottant en un entier non signé en tronquant le résultat
`trunc`	convertit un opérande flottant double précision, respectivement entier de plus grande taille, en un flottant simple précision, respectivement entier de plus petite taille
`float`	convertit un entier en flottant

Instructions arithmétiques

Certaines opérations, comme le modulo, ne sont pas présentes. En effet, le compilateur en connaît le comportement et peut l'émuler à l'aide des autres opérateurs élémentaires.

nom	**signification**
`add`	addition signée
`sub`	soustraction signée
`mul`	multiplication signée
`div`	multiplication signée
`neg`	négation, qui ne marche que sur les types `char`, `short` et `int`

F.5 Les pseudo-instructions liées à la mise au point

Le but de GENVIEW étant notament de fournir des possibilités d'exécution en pas à pas et de visualisation des variables d'un programme, il est demandé au compilateur de produire un code assembleur enrichi des informations nécessaires à la mise au point, en plus des directives d'assemblages classiques. L'usage fait de ces pseudo-instructions découle de l'étude des sources de **gcc**, de morceaux de code générés, et du fichier d'entête `stab.h`, car il n'y a pas de document officiel décrivant leur usage.

nom	**signification**
`.stabd`	numéro de ligne du source pour les instructions suivantes jusqu'au prochain `.stabd`.
`.stabn`	marqueurs de debut et fin de bloc **C**.
`.stabs`	définition d'un type, d'une fonction ou d'une variable avec tous les renseignement concernant sa portée, sa taille, et les types dont il est issu.

La grammaire utile pour l'interpréteur de GENVIEW est donnée dans le tableau ci-dessous. Y sont indiquées les seules valeurs fournies par le compilateur, et donc les seules dont le sens est bien connu.

`.stabd`	`68, 0,` *numéro de ligne*	C'est la seule forme sous laquelle est présente cette instruction. La valeur `68` correspond à la définition de `N_SLINE`, laissant supposer que sont utilisation est plus vaste que le seule numérotage des lignes avec d'autres compilateurs.
`.stabn`	*entier* `, 0, 0,` *étiquette*	Indique les debuts et fins de blocs **C**. Si *entier* vaut `192` (`N_LBRAC`), alors, c'est qu'il y a un début de bloc, c'est à dire un { suivi d'une déclaration de variable dans le corps d'une fonction. `224` (`N_RBRAC`) indique la fermeture de ce bloc. Les variables déclarées entre ces deux pseudo-opérations ne vivent que pendant l'exécution du bloc et obscurcissent les variables de même nom qui ont été déclarées avant. GENVIEW ne sait pas gérer la trace de ces variables — `dbx` non plus —.

`.stabs`	`"`*chaîne*`", 0, 0,` *adresse*	C'est grâce à cette instruction que l'on peut accéder à l'adresse mémoire qui contient une variable. L'*adresse* est soit une étiquette, soit un déplacement dans la pile ou le pointeur de contexte, suivant le contenu de la chaîne. Elle contient : • le nom de la variable, ou le nom d'un nouveau type. • une définition du type, éventuellement en fonction de types complexes précedemment définis. • une définition de ses membres lorsque c'est une structure. • son rayon d'action, global, local, paramètre, statique.

F.6 Exemple de programme

La classique factorielle, prouvant les capacités recursive de la machine, est présentée. Tout d'abord le source **C**.

```
unsigned int
   factorielle(n)
unsigned int n;
{
   return n < 1 ? 1 : n * factorielle(n - 1);
}
```

Puis le résultat de la compilation :

```
gcc_compiled.:
   .stabs "a.c",100,0,0,Ltext
Ltext:
.stabs "int:t1=r1;-2147483648;2147483647;",128,0,0,0
.stabs "char:t2=r2;0;127;",128,0,0,0
.stabs "long int:t3=r1;-2147483648;2147483647;",128,0,0,0
.stabs "unsigned int:t4=r1;0;-1;",128,0,0,0
.stabs "long unsigned int:t5=r1;0;-1;",128,0,0,0
.stabs "short int:t6=r1;-32768;32767;",128,0,0,0
.stabs "long long int:t7=r1;0;-1;",128,0,0,0
.stabs "short unsigned int:t8=r1;0;65535;",128,0,0,0
.stabs "long long unsigned int:t9=r1;0;-1;",128,0,0,0
.stabs "signed char:t10=r1;-128;127;",128,0,0,0
.stabs "unsigned char:t11=r1;0;255;",128,0,0,0
.stabs "float:t12=r1;4;0;",128,0,0,0
.stabs "double:t13=r1;8;0;",128,0,0,0
.stabs "long double:t14=r1;8;0;",128,0,0,0
.stabs "void:t15=15",128,0,0,0

   .text
   .align 4
   .global _factorielle
```

Les premières lignes de `.stabs` définissent les types de base qui vont être utilisé dans le binaire, ainsi que leurs bornes. Ceci est généré par défaut par le compilateur, et Genview n'en tient aucunement compte.

`_factorielle:`	définition du symbole
`  .stabd 68,0,3`	ligne numéro 3
`  sub.si d31,d31,#4`	préparation à un empilement
`  mov.si (d31),d30`	sauvegarde du pointeur de contexte dans la pile
`  mov.si d30,d31`	le pointeur de contexte pointe dans la pile
`  .stabd 68,0,5`	ligne numéro 5
`  cmp.si 8(d30),#0`	comparaison du paramètre a zéro
`  bne #.L2`	y est il égal?
`  mov.si d25,#1`	la valeur de retour de la fonction est 1
`  bra #.L3`	retour de fonction
`.L2:`	
`  sub.si d31,d31,#4`	préparation à un empilement
`  add.si d0,8(d30),#-1`	décrémentation du paramètre
`  mov.si (d31),d0`	empilement du paramètre
`  call $_factorielle`	appel recursif
`  add.si d31,d31,#4`	dépilement pour restorer le contexte
`  mov.si d0,d25`	changement de registre!
`  mul.si d0,d0,8(d30)`	multiplication du paramètre avec la valeur de retour de la fonctions
`  mov.si d25,d0`	re-changement de registre!
`.L3:`	
`  bra #.L1`	
`  .stabd 68,0,6`	ligne numéro 6
`.L1:`	
`  mov.si d30,(d31)`	récupération du pointeur de contexte dans la pile
`  add.si d31,d31,#4`	dépilement de celui-ci
`  ret`	retour à l'appelant
`.stabs "factorielle: F4",36,0,0, _factorielle`	symbole de fonction de type `unsigned int`, d'adresse `_factorielle`
`.stabs "n: p4",160,0,0,8`	symbole de paramètre de type `unsigned int`, d'adresse `8` par rapport à l'adresse `_factorielle`

On peut constater que le compilateur ne fait aucune optimisation, car il y a des mouvements de registres injustifiés, des branchement qui sautent directement à d'autres branchements, des additions avec des nombres négatifs au lieu de soustractions, ne serait-ce que dans ce petit exemple. Néanmoins, cela est strictement sans importance, car cela n'intervient pas sur la mise au point au niveau source, qui reste notre principal cible.

F.7 Fonctions de la librairie standard de GENVIEW

La liste exhaustive — aisément extensible — des fonctions connues par défaut de l'outil est présentée.

```
atoi
strlen
strcmp
printf
sprintf
NAME
DEF_PHFIG
DEF_AB
DEF_PHINS
PHCON
PHREF
PLACE
PLACE_BOTTOM
PLACE_LEFT
PLACE_REF
PLACE_RIGHT
PLACE_TOP
PLACE_ON
COPY_UP_ALL_CON
COPY_UP_CON
COPY_UP_CON_FACE
COPY_UP_ALL_REF
COPY_UP_REF
PHSEG
COPY_UP_SEG
THRU_H
THRU_V
THRU_CON_H
THRU_CON_V
HEIGHT
WIDTH
GET_REF_X
GET_REF_Y
GET_INS_X
GET_INS_Y
GET_CON_X
GET_CON_Y
WIRE1
WIRE2
WIRE3
PHVIA
PLACE_VIA_REF
PLACE_SEG_REF
PLACE_CON_REF
FLATTEN_PHFIG
```

Annexe G

Schémas structurels des différentes *ROM*s

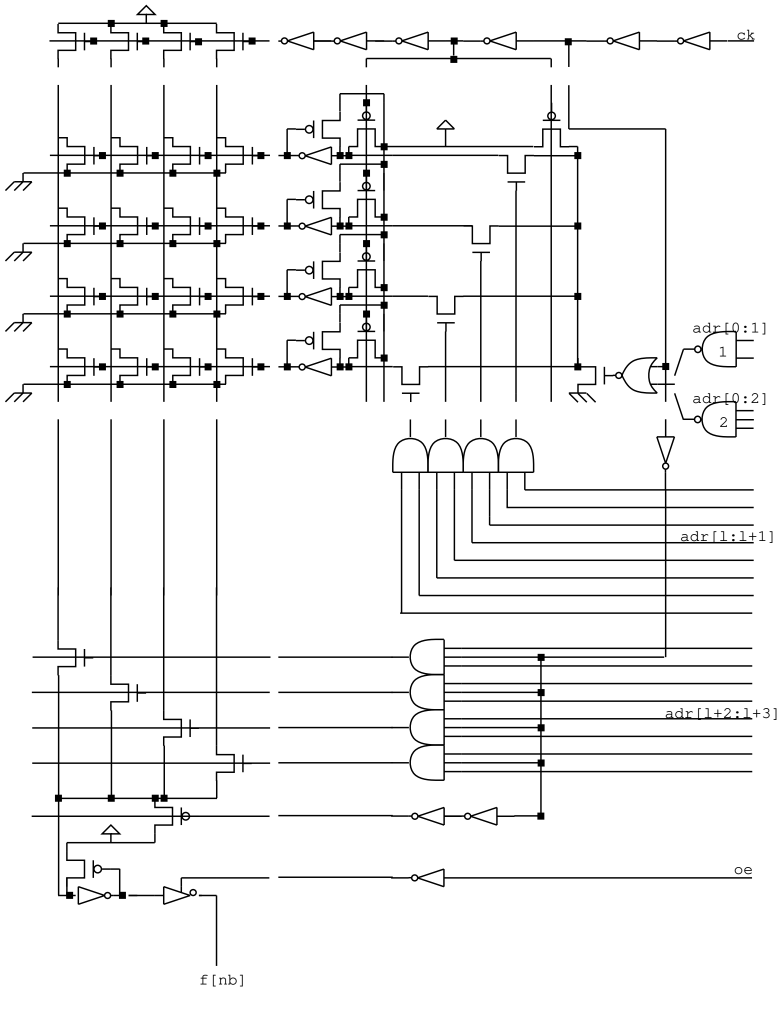


Figure G.1: *ROM*s de 64 et 128 mots.

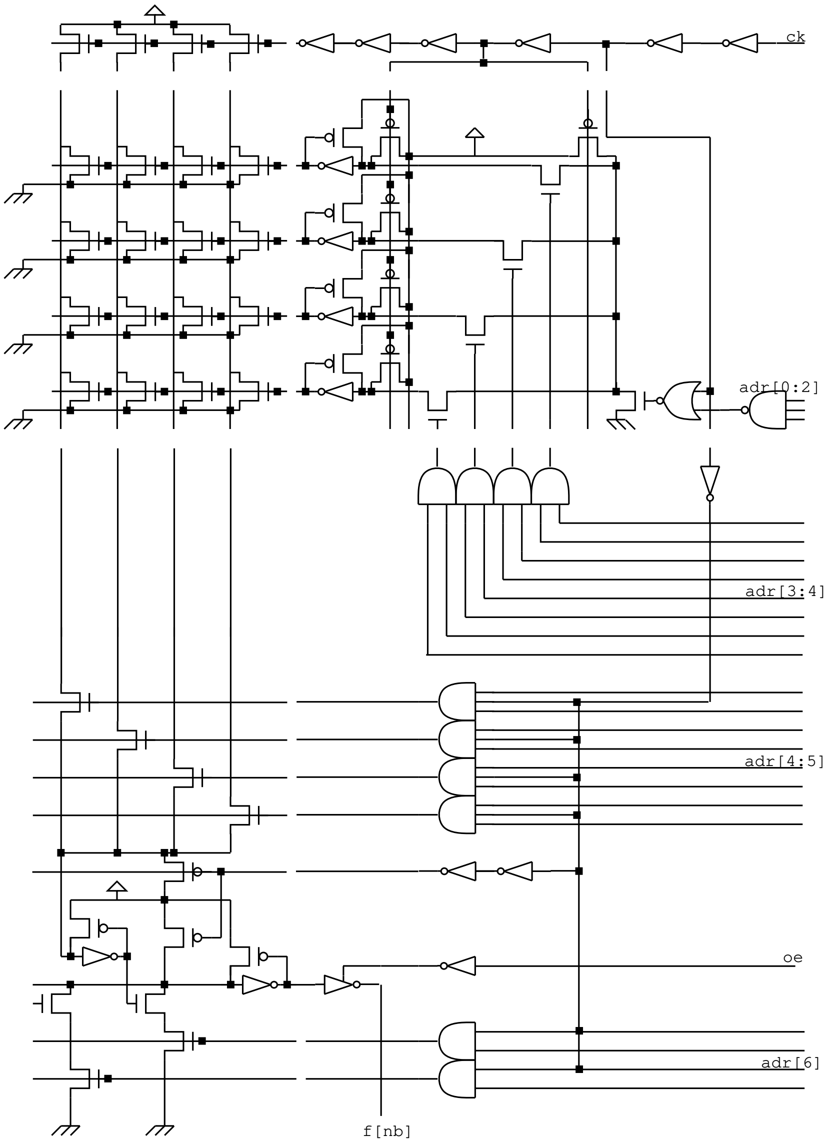


Figure G.2: *ROMs* de 256 mots.

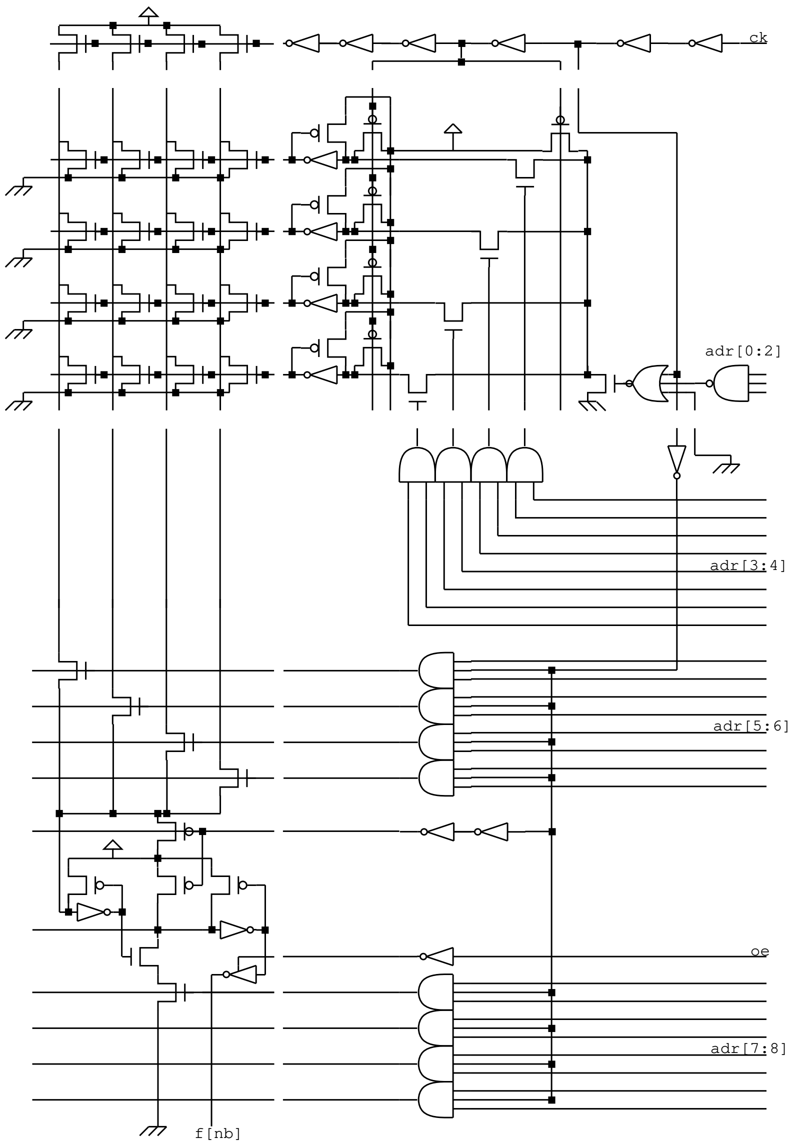


Figure G.3: *ROM*s de 512 mots.

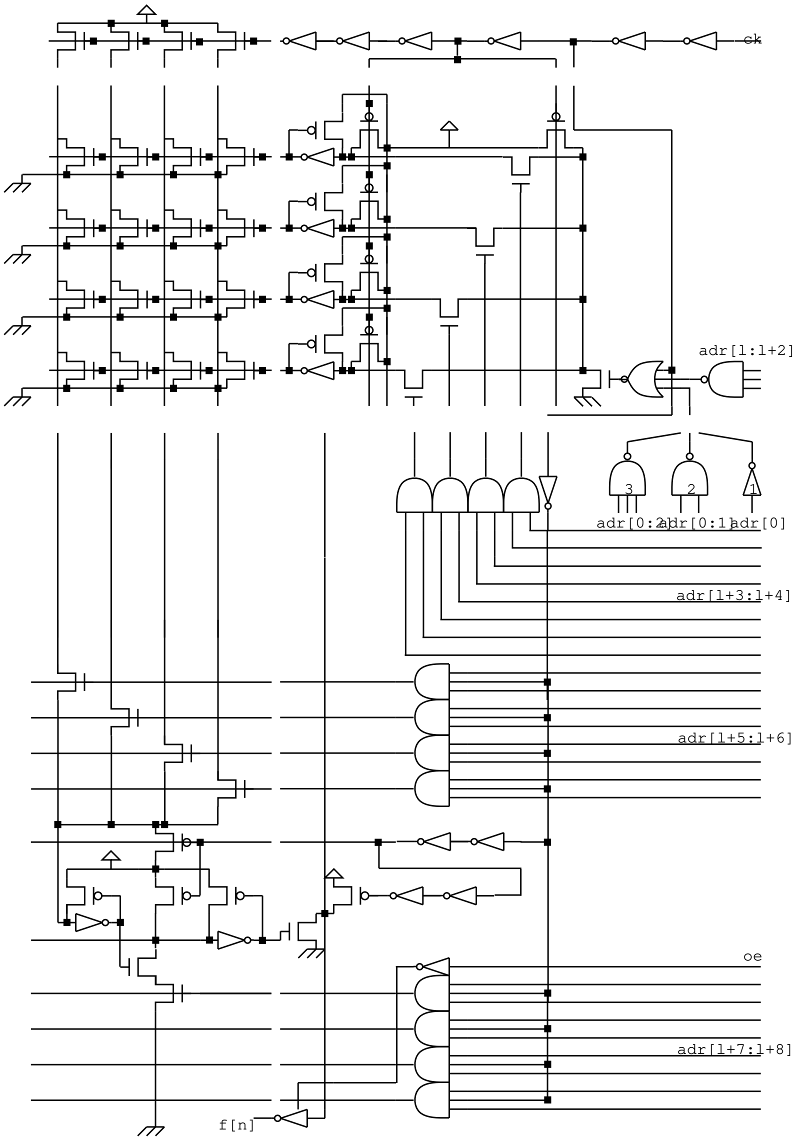


Figure G.4: *ROM*s de 1024 mots et plus.

Annexe H

Makefile de validation d'une *ROM*

La procédure automatisée de validation d'un bloc généré est fournie ci-dessous. Les outils utilisés sont les outils de la chaîne **Alliance**, et ce fichier pourrait être utilisé pour la validation de n'importe quel bloc sans modifications profondes. L'utilisation de l'utilitaire `make` permet de garantir que des modifications sur les objets générés engendrent une phase de vérification.

```
#
SHELL = /bin/sh
NW = 1024
NB = 27
CODE = data
VIEWS = layout vhdl r pat
NAME = rom$(NB)_$(NW)
#tools definition
GROG        = time grog
DRC         = time druc -f
EXTRACTOR   = time lynx -c -t
ABSTRACTOR  = time desb -v
SIMULATOR   = time asimut -b

#make usage       : some indications on how to use this Makefile
usage : Makefile
   @grep make Makefile | grep -v grep | tr -d [#]

#make views       : generates a set of views for a given rom
views $(NAME).cp :
   MBK_CATA_LIB=/users/outil1/grog/dev/cleanup; \
   export MBK_CATA_LIB; \
   $(GROG) $(NB) $(NW) $(CODE) vhdl name=$(NAME)_code; \
   $(GROG) $(NB) $(NW) $(NAME)_code $(VIEWS) name=$(NAME)

#make drc         : verify the layout design rules
drc : $(NAME).cp
   $(DRC) $(NAME)

#make extraction  : extracts the layout and give a transistor netlist
extraction $(NAME).fne : $(NAME).cp
   MBK_OUT_LO=fne; \
   export MBK_OUT_LO; \
   MBK_CATA_LIB=/users/outil1/grog/dev/cleanup; \
   export MBK_CATA_LIB; \
```

```
   $(EXTRACTOR) $(NAME)

#make abstraction : convert the transistor netlist in boolean equations
abstraction $(NAME)_a.vbe : $(NAME).fne
   cp $(NAME).fne $(NAME)_a.fne
   MBK_CATA_LIB=/users/outil1/grog/dev/cleanup; \
   export MBK_CATA_LIB; \
   MBK_IN_LO=fne; \
   export MBK_IN_LO; \
   $(ABSTRACTOR) $(NAME)_a

#make simulation  : simulate logical equations on both extracted and generated
simulation : $(NAME)_a.vbe $(NAME).vbe
   VH_LIBLST=.; \
   export VH_LIBLST; \
   VH_BEHSFX=vbe; \
   export VH_BEHSFX; \
   MBK_CATA_LIB=/users/outil1/grog/dev/cleanup; \
   export MBK_CATA_LIB; \
   $(SIMULATOR) $(NAME) $(NAME) 1 > /dev/null; \
   $(SIMULATOR) $(NAME)_a $(NAME) 2 > /dev/null; \
   grep -v date 1.pat > s1.pat; \
   grep -v date 2.pat > s2.pat; \
   \rm 1.pat 2.pat; \
   diff s1.pat s2.pat
   \rm s1.pat s2.pat $(NAME)*

#make all         : redo the entire verification loop
all : views drc extraction abstraction simulation
```

Bibliographie

[AG92] Lotfi Ben Ammar and Alain Greiner. Automatic placement for bit sliced datapaths. In *ICM 92 proceedings*, pages 4.1.1.1–4.1.1.4, December 1992.

[AG93] Lotfi Ben Ammar and Alain Greiner. A high density datapath compiler mixing random logic with optimized blocks. In *The European Conference on Design Automation*, pages 294–298, February 1993.

[AGR70] S.B. Ackers, J.M. Geyer, and D.L. Roberts. IC mask layout with a single conductor layer. In *7th Design Automation Conference*, pages 7–16, 1970.

[ANS89] *American National Standard for Information Systems—Programming Language C*, 1989. X3.159.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques and tools*. Addison-Wesley, 1986.

[AW83] Bryan Ackland and Neil Weste. An automatic assembly tool for virtual grid symbolic layout. In *VLSI' 83*, pages 457–465. Elseiver Science Publishers B.V, Inc, 1983.

[BCdV91] Luc Burgun, Damien Cavaillé, and Franck de Vargas. C4, a complex cmos cell compiler. Technical report, Laboratoire MASI, Équipe cao-vlsi, June 1991.

[BH80] J. Batali and A. Hartheimer. The design procedural language. Technical Report 598, MIT, September 1980.

[Boy88] D. Boyer. Symbolic layout compaction review. In *25th Design Automation Conference*, pages 383–389, 1988.

[Bur88] M.R. Burich. Design of module generators and silicon compiler. In D. D. Gajski, editor, *Silicon Compilation*, chapter 2, pages 49–94. Addison Wesley, 1988.

[BV92] Cyrus S. Bamji and Ravi Varadarajan. Hierarchical pitchmatching compaction using minimum design. In *29th Design Automation Conference*, pages 311–317, 1992.

[BV93] Cyrus S. Bamji and Ravi Varadarajan. MTSC: A method for identifying overconstraints during hierarchical compaction. In *30th Design Automation Conference*, pages 389–394, 1993.

[BV94] Pirouz Bazargan Sabet and Huu Nghia Vuong. Tradeoffs in designing a VHDL simulator. In *26th Southeastern Symposium on System Theory*, pages 362–367, Athens, Ohio, March 1994.

[Cad91a] Cadence Design Systems, Inc, 555 River Oaks Parkway, San Jose, California 95134. *ModuleMaker User Guide*, October 1991.

[Cad91b] Cadence Design Systems, Inc, 555 River Oaks Parkway, San Jose, California 95134. *SKILL Reference Manual*, October 1991.

[Cas91] Andrea Casotto. *Octtools 5.1: Part 2, Reference*. Electronics Research Laboratory, University of California, Berkeley, October 1991.

[Cas93] Cascade Design Automation Corporation, 3650 131st Avenue SE, Bellevue, WA 98006. *CDS Users's Manual and CDS Reference Manual*, April 1993.

[CFJ93] Jacques Chambon, Gilles Fourneris, and Robert Jamier. A system and a methodology for the development of technology independent parameterized module generators. Submited paper, 1993.

[Cha92] Philippe Chaisemartin. Comparaison and evaluation of different datapath generators. Thomson ST Internal technical report, June 1992.

[Cla90] J.L. Clark. Summary of features available within MGL. Technical report, Plessey Semiconductors Ltd, GEC Plessey Semiconductors Ltd, Cheny Manor, Swindon, Wiltshire, October 1990.

[CM88] E.K. Cheng and S. Mazor. The genesil silicon compiler. In Daniel D. Gajski, editor, *Silicon Compilation*, chapter 9, pages 361–404. Addison Wesley, 1988.

[Com91] Compass Design Automation. *VIP Language*, v8r3 edition, 1991.

[CS88] Vince Corbin and Warren Snapp. Design methodology of the concorde silicon compiler. In Daniel D. Gajski, editor, *Silicon Compilation*, chapter 10, pages 406–445. Addison Wesley, 1988.

[DMH$^+$93] Joseph Dao, Nobu Matsumoto, Tsuneo Hamai, Chusei Ogawa, and Shojiro Mori. A compaction method for full VLSI layout. In *30th Design Automation Conference*, pages 407–412, 1993.

[DNJ90] Jean-Claude Dufourd, Jean-François Naviner, and Francis Jutand. Preform, a process independant symbolic layout system. In *ICCAD-90*, pages 248–251, Santa Clara, California, November 1990.

[Dun80] A.E. Dunlop. SLIM, the translation of symbols into mask data. In *17th Design Automation Conference*, pages 595–608, June 1980.

[EDI87] EDIF Steering Committee Electronic Industries Association. *Electronic Design Interchange Format, version 2.0*, 1987.

[Fra92] Jean-Arnaud François. *F-RISC: Machine risc modulaire intégrant une unité flottante*. PhD thesis, Laboratoire cao-vlsi, 4, place Jussieu, 75252 Paris cedex 05, France, 1992.

[Gaj88] Daniel D. Gajski, editor. *Silicon Compilation*. Addison-Wesley, 1988.

[GDWL92] Daniel D. Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *High Level Synthesis, Introduction to Chip and System Design*, chapter 1, pages 18–19. Kuwer Academic Publisher, 1992.

[GL90] Alain Greiner and Jean-Pierre Leroy. A symbolic layout view in edif for process independant design. In *Fourth European Edif Forum proceedings*, Daresbury, Cheshire, UK, 1990.

[GLWW94] Alain Greiner, Luis Lucas, Franck Wajsbürt, and Laurent Winckel. Design of a high complexity superscalar microprocessor with the portable IDPS ASIC library. In *The European Conference on Design Automation*, March 1994.

[GN76] D. Gibson and S. Nance. Slic, symbolic layout of integrated circuits. In *13th Design Automation Conference*, pages 434–440, 1976.

[GP92] Eric Gautrin and Laurent Perraudeau. MADMACS : A tool for the layout of regular arrays. Rapport de recherche 1670, INRIA, May 1992.

[GP94] Alain Greiner and Frédéric Pétrot. A public domain high performances portable ROM generator. In *20th Euromicro*, pages 414–419. IEEE, September 1994.

[HMGK91] Amjad Hajjar, Roland Marbot, Alain Greiner, and Payam Kiani. TAS: An accurate timing analyzer for CMOS VLSI. In *The European Conference on Design Automation*, pages 261–265, Paris, France, February 1991.

[IDP90] IDPS. Very fast implementation of complex systems on silicon. Technical annex 3.2, IDPS, European Community Esprit project, May 1990. Jean-Pierre Moreau, Editor.

[Jan91] Henry Janssen. *Modgen and graphMG User Manual*. Philips Research Laboratories, Eindhoven, The Netherlands, 1991. Modgen has been chosen by the IDPS ESPRIT project as generator exchange language.

[JM84] S.C. Johnson and S. Mazor. Silicon compiler let system makers design their own VLSI chips. *Electronic Design*, 32(20): 167–181, October 1984.

[Joh75] S.C. Johnson. Yacc - yet another compiler-compiler. Computer science technical report no. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.

[Joh79] D. Johannsen. Bristel blocks : A silicon compiler. In *16th Design Automation Conference*, 1979.

[Kin85] Chris Kingsley. *v5 Composition File Format*. VTI Design Technology Group, 1865 Lundy Avenue, San Jose, CA 95131, August 1985.

[Lar71] R.P. Larsen. Computer aided preliminary design of customized MOS arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 20: 512–523, May 1971.

[Les75] M.E. Lesk. Lex - a lexical analyzer generator. Comp. sci. tech. rep. no. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.

[LGT93] Luis Lucas, Alain Greiner, and Michel Thill. High speed adder generator. In *5th International Conference on Microelectronic*, pages 136–139, December 1993.

[LL80] A.D. Lopez and H-F.S. Law. A dense gate matrix layout method for MOS VLSI. *IEEE Trans on E.D*, 27(8), August 1980.

[LRS+82] R.J. Lipton, S.C. North R., Sedgwick, J. Valdes, and G. Vijayan. ALI: A procedural language to describe VLSI layouts. In *19th Design Automation Conference*, pages 467–473, 1982.

[LW83] Y.Z. Liao and C.K. Wong. An algorithm to compact VLSI symbolic layouts with mixed constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 2.2: 62–69, April 1983.

[Mar83] Aaron Marcus. Graphic design for computer graphics. *IEEE Computer Graphics and Applications*, 3(7): 63–82, July 1983.

[Mar90] D. Marple. A hierarchial preserving hierarchical compactor. In *27th Design Automation Conference*, pages 375–381, 1990.

[MC80] Carver Mead and Lin Conway. *Introduction to VLSI Systems*. Addison Welsey, 1980.

[McG93] Philip McGee. The asa silicon compiler library development system. In *The European Conference on Design Automation, User Forum and EUROASIC Prizes*, pages 6–10, 1993.

[Men90] Mentor Graphics Corporation, 2045 Hamilton Avenue, San Jose, CA 95125. *GDT Designer, L Database and Language Users Guide*, September 1990.

[MSH90] David Marple, Michiel Smulders, and Henk Hegen. Tailor: A layout system based on trapezoidal corner stitching. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 9(1): 66–90, January 1990.

[Nag75] L.W. Nagel. SPICE2: A computer program to simulate semiconductor circuits. Technical Report ERL-M520, University of California, Berkeley, May 1975.

[Nav92] Jean-François Naviner. *Préforme : étude et développement d'une méthode de conception symbolique de cellules indépendantes de la technologie*. PhD thesis, Télécom Paris, 1992.

[NS79] W.M. Newman and R.F. Sproull. *Principles of Interactive Computer Graphics*. Mc Graw - Hill, 2nd ed. edition, 1979.

[Nye88] Adrian Nye. *Xlib Programming Manual*. O'reilly and Associates, Inc., 1988.

[Pét92] Frédéric Pétrot. *Genlib procedural generation functions*. Laboratoire MASI, Équipe cao-vlsi, 4, place Jussieu, 75252 Paris cedex 05, France, 1992.

[PSF93] V. Patel, M. Shah, and G. Floyd. Report on ROM generator implementation. Esprit 5075: IDPS - WP2 cell libraries, GEC Plessey Semiconductors, May 1993.

[RW82] J. Rosenberg and N. Weste. The ABCD language. Technical Report 82-01, Microelectronic Center of North Carolina, 1982.

[S+89] Richard Stallman et al. Gnu gcc c compiler version 1.42. 675 Mass avenue, Cambridge, MA 02139, USA, 1989.

[Sch85] Jean-Pierre Schoellkopf. *Siliciel, Contributions à l'architecture des circuits intégrés et à la compilation du silicium*. PhD thesis, Institut National Polytechnique de Grenoble, April 1985.

[Sch87] A.F. Schwartz. *Computer Aided Design of Microelectronic circuits and systems*. Academic Press, 1987.

[Shu85] Bob Shur. *HNL format and Programming Tools*. VTI Design Technology Group, 1865 Lundy Avenue, San Jose, CA 95131, February 1985.

[Ste77] Roger G. Stewart. High density CMOS ROM arrays. *IEEE Journal of Solid State Circuits*, 12: 502–506, October 1977.

[SW85] Larry Saunders and Ronald Waxman. *The VHDL Language Reference Manual*. IEEE, Inc, 345 East 47th Street, New York, NY 10017, USA, 1985.

[Ute91] M.C. Utesch. A new approach to hierarchical adaptation using sequence control based on cell interactions. In *28th Design Automation Conference*, pages 723–726, June 1991.

[WDJ79] C.R. Wilcox, M.L. Dageforde, and G.A. Jirak. *Mainsail Language Manual*. Xidak, Inc, Los Altos, CA, August 1979.

[WE85] N. H. Weste and K. Eshraghian. *Principles of CMOS VLSI Design, A System Perspective*. Addison-Wesley, 1985.

[Wer83] J. Werner. Progress toward the "ideal" silicon compiler. *VLSI desing*, September 1983.

[WL86] G. Wood and H-F.S. Law. Skill - an interactive procedural design environment. In *IEEE Custom Integrated Circuits Conference*, pages 544–547, Portland, OR, May 1986. IEEE, Inc.

[YCD+93] So-Zen Yao, Chung-Kung Cheng, Debaprosad Dutt, Surendra Nahar, and Chi-Yuan Lo. Cell-based hierarchical pitchmatching compaction using minimal LP. In *30th Design Automation Conference*, pages 395–400, San Diego, CA, 1993.

[Ze91] James S. Medou Zengue Ze. *Vérification automatique des règles de dessin des circuits vlsi: Règles formelles, Approche hiérarchique*. PhD thesis, Université Pierre et Marie Curie, Laboratoire MASI, October 1991.

Abstract

This thesis introduces the concept of 'portable library' and details a dedicated environment oriented toward the development of such layout libraries.

The first chapter presents a thin fixed grid symbolic layout methodology that ensure process independence on a wide range of technologies. This approach warranties the respect of pitch matching constraints between unrelated cells location for abutment based custom blocks. Chapter two presents canonical data structures for the representation of the symbolic layout and structural views of *VLSI* circuits. All the tools of the **Alliance** CAD system working on the physical or logical views are based on these data structures. The third chapter presents a language dedicated to the procedural generation of custom blocks designed using a specific layout leaf cell library and a tiling program. This language is in fact a set of **C** functions that heavily relies on the data structures. In order to help the debug of procedural generators by designers unaware of software engeenering practices, we introduce in chapter four a graphical tool that simplifies this task. It contains a text editor, a **C** compiler targetted to a virtual machine, an interpreter with symbolic debugging facilities, and a layout visualisation tool. Finally, the general design methodology for the generators using a *tiler and leaf cell* approach is illustrated through the example of a Read Only Memory.

Keywords
process independence
canonical data structures
procedurale generation
interpretation, debug