[REB97] V. REBALDI, "Conception et réalisation d'un routeur de paquets à hautes performances", Thèse de doctorat de l'université Paris VI, Paris, France, 1997

[ROH95] E. ROHM, "Latest Benchmarks Results of VHDL Simulation Systems", Proc. EURO-DAC '95, pp 406-411, Brighton, Great Britain, September 18-22, 1995

[SAR95] C. SARWARY, "Synthèse d'automates d'états fini pour les circuits intégrés VLSI : Optimisation du codage par évaluation du coût après synthèse et synthèse d'automates à pile", Thèse de doctorat de l'université Paris VI, Paris, France, 1995

[SHA38] C.E. SHANNON, "A Symbolic Analysis of Relay and Switching Circuits", Trans. AIEE, vol. 57, pp 713-723, 1938

[SHA86] M. SHAHDAD, "AN OVERVIEW OF VHDL LANGUAGE AND TECHNOLOGY", Proc. 23rd DAC, pp 320-326, 1986

[SIA94] Semiconductor Industry Association. The National Technology Roadmap for Semiconductors. San Jose, California, USA, 1994

[SIM98] Encyclopedia of Computer Science, Simulation Article, 4th Edition, 1998, IEEE press (To be published)

[SIN91] S. SINGHANI, "Code Generation for VHDL – Interpreted vs. Compiled Code", Proc. EUROVHDL 1991, pp 336-339

[SMI87] S. SMITH, M. RAY MERCER, B. BROCK, "Demand Driven Simulation : BACKSIM", Proc. 24th DAC, ACM/IEEE, pp 181-187, 1987

[TAB94] M. TABUSSE, "Démarrez votre conception d'Asic en langage comportemental], Electronique Magazine, n°35, pp 41-50, Février 1994

[THO91] D.E. THOMAS, P.R. MOORBY, "The Verilog Hardware Description Language", Kluwer Academic Publishers, Boston, 1991.

[USA89] K. USAMI, S. IWAMURA, "Optimized design Method for full custom microprocessors", Proc. IEEE, Custom Integrated Circuits Conference, 1989.

[WAN90] Z. WANG, P. MAURER, "LECSIM : A LEVELIZED EVENT DRIVEN COMPILED LOGIC SIMULATOR", Proc. 27th DAC, ACM/IEEE, pp. 491-496, 1990

[WES94] N.H.E. WESTE, K. ESHRAGHIAN, "Principles of CMOS VLSI design : a systems perspective", 2nd edition, Addison Wesley Pub., ISBN 0-201-53376-6, Avril 1994

[CHI67] J. CHINAL, "Techniques Booléennes et Calculateurs Arithmétiques", Ed. DUNOD, Paris, France, 1967

[DEW84] A. DEWEY "VHDL : IMPLICATIONS OF A MODERN HARDWARE DESCRIPTION LANGUAGE", Proc. ICCD 1984, pp 78-81, 1984

[FLY97] M.J. FLYNN, "What's Ahead in Computer Design ?", Euromicro 97, Proceedings of the 23rd Euromicro Conference, pp 4-9, Budapest, Hungary, September 1-4, 1997

[GAU87], M.C. GAUDEL, M. SORIA, C. FROIDEVAUX, "Types de données et algorithmes", "Recherche, Tri, Algorithme sur les graphes, volume 2", EDITION DIDACTIQUE, INRIA, France, 1987

[GIR85] J.J. GIRARDOT, "Les langages et les systèmes LISP", éditests, France, 1985

[GRE93] A. GREINER, F. PECHEUX, "Alliance : A complete set of CAD tools for teaching VLSI design", Proc. 3rd eurochip workshop on VLSI design training, PP 230-237, 1993

[GRE94] A. GREINER, L. LUCAS, F. WAJSBURT, L. WINCKEL, "Design of a High Complexity Superscalar Microprocessor with the Portable IDPS ASIC Library", Proc. EDAC 94, Paris, FRANCE

[HAJ91] A. HAJJAR, A. GREINER, R. MARBOT, P. KIANI "TAS : An Accurate Timing Analyser for CMOS VLSI", Proc. EDAC 91, pp 261-265, Amsterdam, The Netherlands, February 1991.

[HAJ92] A. HAJJAR, "Modélisation des temps de propagation et analyse temporelle statique des circuits intégrés CMOS", Thèse de doctorat de l'université Paris 6, Juillet 1992.

[HUE91] M. HUEBER, S. LASSERRE, A. de MONTEVILLE, "VHDL EXPERIMENTS ON PERFORMANCE", Proc. Euro-VHDL, pp 282-292, 1991

[IEEE88] "IEEE standard VHDL Language Reference Manual", IEEE, New York, USA, 1988

[LOU88] M. LOUGHZAIL, M. COTE, M. ABOULHAMID, E. CERNY "EXPERIENCE WITH THE VHDL ENVIRONMENT", Proc. 25th DAC, pp 28-32, 1988

[LUC95] L. LUCAS, "Conception et réalisation d'un microprocesseur VLIW : méthodologie de conception et implantation VLSI", Thèse de doctorat de l'université Paris 6, Décembre 1995

[MAU90] P. MAURER, Z. WANG, "TECHNIQUES FOR UNIT-DELAY COMPILED SIMULATION", Proc. 27th DAC, ACM/IEEE, pp 480-484, 1990

[MEA83] C. MEAD, L. CONWAY, "INTRODUCTION AUX SYSTEMES VLSI", Edition Française, Interéditions, Paris, France, 1983

[MOR82] B.M.E. MORET, "Decision Trees and Diagrams", Computing Surveys, Vol. 14, pp 593-623, Decembre 1982

[NAG75] L.W. NAGEL, "SPICE2: A computer program to simulate semiconductor circuits", Memo. ERL-M250, Electronics Research Lab., Univ. California, Berkeley, July 1978

[PET94] F. PETROT, "Outils d'aide au développement de bibliothèques VLSI portables", Thèse de doctorat de l'université Paris 6, Juillet 94.

ARTICLES PRESENTES DANS DES CONFERENCES INTERNATIONALES

[BAZ93] P. BAZARGAN SABET, A. GREINER, H.N. VUONG "ASIMUT : A public domain VHDL simulation tool", Proc. 4th Eurochip Workshop, PP 62-65, 29th September-1st October 1993, Toledo, Spain

[BAZ94] P. BAZARGAN SABET, H.N. VUONG, "Trade-Offs in Designing a VHDL Simulation Tool for Digital VLSI", Proc. 26th Southeastern Symposium on System Theory, IEEE Computer Society, PP 380-384, March 20-22, 1994, Athens, Ohio, USA

[BAZ95] P. BAZARGAN SABET, H.N. VUONG, "Experiments with a Modular VHDL simulation Tool for Digital VLSI", Proc. European Simulation Multiconference 1995, SCS, PP 748-752, June 5-7, 1995, Prague, Czech Republic.

ARTICLES UTILISES POUR LE MANUSCRIT

[AIR90] R. AIRIAU, J.M. BERGE, V. OLIVE, J. ROUILLARD "VHDL, du langage à la modélisation", PRESSES TECHNIQUES ET UNIVERSITAIRES ROMANDES, Lausanne, Suisse, 1990

[AKE78] S.B.AKERS, "Binary Decision Diagrams", IEEE Trans. On Computers, vol. c-27, N°6, June 1978, USA

[ALL78] J. ALLEN, "Anatomy of Lisp", Mc GRAW-HILL BOOK COMPANY, USA, 1978

[AMA91] A. AMARA, P. BAZARGAN SABET, J. BENZAKKI, M. ISRAEL, "A VHDL Subset for Optimal Simulation : Application in Procedural Generators", Proc. of 9th IASTED International Symposium, Innsbruck, Austria, February 1991

[BAZ94] P. BAZARGAN SABET, J. DUNOYER, A. GREINER, M.M. ROSSET LOUERAT "A Tutorial for Advanced VLSI Course : Designing the 32 bit DLX Microprocessor With the ALLIANCE CAD System", Proc. 5th Eurochip Workshop on VLSI Training", pp 120-127, 17, 18, 19 October 1994, Dresden, GERMANY

[BRA90] K. BRACE, R. RUDELL, R. BRYANT, "Efficient implementation of a BDD package", Proc. 27th DAC, pp40-45, 1990

[BRY86] R. BRYANT, "Graph-Based algorithms for Boolean function manipulation", IEEE Trans. On Computers, vol C-35, N° 8, pp 677-691, August 1986

[BRY90] R. BRYANT, "Symbolic Simulation – Techniques and Applications", Proc. 27th DAC, ACM/IEEE, pp 517-521, 1990

[BRY92] R. BRYANT, "Symbolic Boolean Manipulation with Ordered Binary Decisions Diagrams", ACM, Computing Surveys, vol. 24, N°3, Sept. 1992

[BRY94] R. BRYANT, Y. CHEN, "Verification of Arithmetic Functions with Binary Moment Diagrams", 37 pages, Carnegie Mellon University, Pittsburgh, USA, May 1994

[BUR93] L. BURGUN, "Synthèse logique de circuits intégrés VLSI : Utilisation de graphes de décision binaire dans la phase d'optimisation comportementale", Thèse de doctorat de l'université Paris VI, Paris, France, 1993

[CAR93] M. CARROLL, "VHDL – panacea or hype ?", IEEE Spectrum, pp 34-37, June 1993

Bibliographie

6)- Bench2 vectorisé, Chaîne d'inverseurs (description structurelle)

Temps de compilation (en secondes) :

Nombre d'inverseurs 32 bits	Asimut BDD	SYNOPSYS	CADENCE
10	1,20	4,90	31,54
50	0,91	5,07	32,08
100	0,32	5,30	32,92
500	1,41	7,18	41,88
1000	3,35	9,89	58,31
2000	8,45	15,60	114,53
4000	31,77	28,95	358,21

Temps d'exécution (en secondes) :

Nombre d'inverseurs 32 bits	Asimut BDD	SYNOPSYS	CADENCE
10	8,80	17,11	7,12
50	49,09	74,02	33,5
100	99,68	147,16	62,18
500	498,59	929,54	392,65
1000	996,65	2098,53	945,37
2000	1991,55	4354,20	2092,22
4000	3968,23	9653,43	4708,78

Mémoire maximum nécessaire (en Ko) :

Nombre d'inverseurs 32 bits	ASIMUT bdd	SYNOPSYS	CADENCE
10	20201,00	4330,00	4830
50	22806,00	5035,00	5008
100	24158,00	5881,00	5496
500	32600,00	12131,00	8916
1000	42649,00	19995,00	13221
4000	51844,00	33368,00	19636

2504	18564	7696	9354	20700

4)- Rcube

Temps de compilation (en secondes) :

Asimut Bdd	Synopsys	Cadence	Asimut Abl
27,94	17,96	71,74	27,08

Temps d'exécution (en secondes) :

Nombre de cycles	Asimut Bdd	Synopsys	Cadence	Asimut Abl
1259	744	693,32	84,92	838,66
10259	5341,36	5769,53	720,02	6188,96

Mémoire nécessaire à la compilation (en Ko) :

Asimut Bdd	Synopsys	Cadence	Asimut Abl
18999	7718	7468	18737

5)- Bench1 vectorisé, Chaîne d'inverseurs (description comportementale)

Temps de compilation (en secondes) :

Nombre d'inverseurs 32 bits	ASIMUT	SYNOPSYS	CADENCE
10	0,70	0,59	2,74
50	1,01	0,74	3,13
100	0,54	0,82	3,76
500	3,63	2,25	10,14
1000	9,49	4,13	24,19
2000	27,85	9,22	64,93
4000	119,13	19,58	251,89

Temps d'exécution (en secondes) :

Nombre d'inverseurs 32 bits	ABL	SYNOPSYS	CADENCE
10	19,07	17,58	6,87
50	72,51	73,27	27,82
100	140,53	144,21	60,69
500	890,25	977,73	450,45
1000	1861,42	2713,23	1026,36
2000	3943,35	4414,39	2236,23
4000	8849,59	9702,54	4956,28

Mémoire maximum nécessaire (en Ko) :

Nombre d'inverseurs 32 bits	ASIMUT	SYNOPSYS	CADENCE
10	20131,00	4310,00	4492
50	22609,00	4623,00	4937
100	23891,00	5004,00	5346
500	31682,00	8299,00	8278
1000	41181,00	12498,00	11952
4000	49128,00	19132,00	16908

Mémoire nécessaire à la compilation (en Ko) :

Nombre d'inverseurs	Asimut BDD	SYNOPSYS	Cadence	Asimut ABL
10	798,00	1930,00	2543,5	921,00
50	1007,00	2500,00	3007,5	1198,00
100	1034,00	2624,00	3366	1234,00
500	1333,00	4546,00	5764	1330,00
1000	1512,00	6658,00	7747,5	1563,00
1500	1783,00	8805,00	9664	1731,00
2000	1931,00	10971,00	11920	1896,00
4000	2605,00	18830,00	19759	2604,00
10000	4662,00	43179,00	44162	4653,00
20000	8513,00	81490,00	43204	8514,00

Mémoire nécessaire à l'exécution (en Ko) :

Nombre d'inverseurs	Asimut BDD	SYNOPSYS	Cadence	Asimut ABL
10	2801,00	3930,00	2840	2777,00
50	2964,00	4206,00	3839,5	2969,00
100	3076,00	4495,00	4270	3073,00
500	3537,00	5900,00	3217	3523,00
1000	4069,00	7913,00	6835,5	4061,00
1500	4622,00	9633,00	7981	4622,00
2000	5149,00	11251,00	9122	5150,00
4000	7320,00	18292,00	13682	7311,00
10000	13613,00	19300,00	27376	13604,00
20000	24792,00	37228,00	42111	24803,00

3)- MIPS

Temps de compilation (en secondes) :

ASIMUT-BDD	SYNOPSYS	CADENCE	ASIMUT-ABL
7,53	3,88	12,33	5,25

Temps d'exécution (en secondes) :

Nombre de cycles	ASIMUT-BDD	SYNOPSYS	CADENCE	ASIMUT-ABL
726	47,07	14,21	12,8	90,44
1569	102,89	28,27	26,46	195,82
2105	131,88	36,45	33	266,31
2504	157,2	44,29	33,88	315,73

Mémoire de compilation (en Ko) :

ASIMUT-BDD	SYNOPSYS	CADENCE	ASIMUT-GEX
10250	5928	6848	10052

Mémoire d'exécution (en Ko) :

Nombre de cycles	ASIMUT-BDD	SYNOPSYS	CADENCE	ASIMUT-ABL
726	15051	7418	9006	18640
1569	17131	7620	9290	21394
2105	17813	7659	9360	22096

Mémoire nécessaire à l'exécution (en Ko) :

Nombre d'inverseurs	BDD	SYNOPSYS	Cadence	Asimut ABL
10	2653,50	4035,00	2869,00	2659,00
50	2841,00	4166,00	4027,00	2829,00
100	2936,50	4263,00	4316,00	2931,00
500	3260,00	5128,50	5616,00	3213,00
1000	3632,00	6427,00	6664,00	3524,50
1500	4042,00	7404,00	7748,00	3871,50
2000	4395,00	7404,50	7744,00	4183,50
4000	5940,00	13145,50	13048,00	5444,00
10000	10223,00	13472,00	25739,00	9270,00
20000	17679,00	24272,00	23480,00	15776,00

2)- Bench 2, Chaîne d'inverseurs (description structurelle)

Temps de compilation (en secondes) :

Nombre d'inverseurs	ASIMUT bdd	SYNOPSYS	Cadence	Asimut ABL
10	0,95	1,42	2,835	0,85
50	1,05	1,45	3,17	1,00
100	1,45	1,71	3,67	1,25
500	1,05	3,21	8,95	0,56
1000	0,93	5,36	20,205	1,24
1500	1,03	7,31	34,11	1,31
2000	1,61	9,48	55,39	1,31
4000	4,54	21,18	204,47	4,82
10000	20,69	60,52	1172,03	21,77
20000	81,19	247,17	4499,6	79,53

Temps d'exécution (en secondes) :

	ASIMUT bdd	SYNOPSYS	Cadence	Asimut ABL
10	1,36	2,89	1,86	2,13
50	3,71	11,14	2,89	3,87
100	8,22	22,37	4,55	9,20
500	36,49	110,21	19,75	39,24
1000	75,72	238,52	44,98	76,04
1500	113,45	360,33	58,14	116,65
2000	153,83	501,00	78,01	158,40
4000	324,07	1188,53	180,08	370,41
10000	1086,39	3990,57	884,83	1103,22
20000	2961,36	10754,82	2088,87	2972,61

Ces tableaux sont issus des mesures de performance effectuées dans le cadre de la comparaison entre les simulateurs VHDL logiques Asimut/Cadence/Synopsys. Les données les plus pertinentes ont été présentées au chapitre 8.

1)- Bench 1, Chaîne d'inverseurs (description comportementale).

Temps de compilation (en secondes) :

Nombre d'inverseurs	ASIMUT bdd	SYNOPSYS	Cadence	Asimut ABL
10	0,85	0,60	2,81	0,60
50	1,00	0,66	3,06	1,05
100	0,80	0,79	3,39	1,05
500	0,97	2,13	8,92	0,89
1000	1,27	3,22	15,43	0,75
1500	1,67	4,06	26,57	0,68
2000	1,07	4,31	26,74	1,32
4000	3,01	11,68	133,80	2,35
10000	10,40	43,50	969,95	7,67
20000	23,61	129,79	3632,52	18,99

Temps d'exécution (en secondes) :

Nombre d'inverseurs	ASIMUT bdd	SYNOPSYS	Cadence	Asimut ABL
10	1,24	3,49	1,81	1,73
50	3,78	11,06	3,15	3,77
100	7,27	21,80	4,48	7,87
500	37,65	118,50	30,31	37,62
1000	75,51	244,33	48,58	80,46
1500	112,85	381,61	85,37	127,93
2000	155,41	434,03	79,73	169,91
4000	364,26	1245,16	332,45	382,91
10000	1081,99	4141,20	1186,39	1180,84
20000	2980,20	11771,89	3302,45	3188,99

Mémoire nécessaire à la compilation (en Ko) :

Nombre d'inverseurs	BDD	SYNOPSYS	Cadence	Asimut ABL
10	782,50	1834,50	2506,00	865,50
50	750,50	1878,50	2739,00	765,00
100	897,00	2364,00	3149,00	1017,50
500	1330,00	4140,00	4735,00	1268,50
1000	1570,50	5740,50	5903,00	1496,50
1500	1785,00	7586,00	6881,00	1684,00
2000	1949,00	7570,50	6885,00	1836,50
4000	2798,00	15961,00	11190,00	2532,00
10000	4978,00	37422,00	21367,00	4392,00
20000	8777,00	71331,00	22620,00	7669,00

Annexe 4

Tableaux de mesure

1)- Bench 1, Chaîne d'inverseurs (description comportementale).

2)- Bench 2, Chaîne d'inverseurs (description structurelle)

3)- MIPS

4)- Rcube

5)- Bench1 vectorisé, Chaîne d'inverseurs (description comportementale)

6)- Bench2 vectorisé, Chaîne d'inverseurs (description structurelle)

```
  unsigned char *VALU;                 /* pointer on the value          */
  }
redlst_list;
```

- **Valu** : Pointeur sur la valeur courante du signal.
- **Next** : Elément suivant de la liste.

1.9) Valeurs projetés des signaux gardés : PRJBVL, PRJRVL

Ces structures contiennent les bi-valeurs projetées qui représentent la valeur des émetteurs et la valeur de leur condition de connexion/déconnexion.

```
typedef struct prjbvl           /* projected bi-value                   */
  {
  struct prjbvl *NEXT  ;        /* next bi-value (local)                */
  struct prjbvl *GLNEXT;        /* next bi-value (global)               */
  unsigned char  CNDVAL;        /* driver's condition value             */
  unsigned char  DRVVAL;        /* driven value                         */
  }
prjbvl_list;

typedef struct prjrvl           /* projected bi-value                   */
  {
  struct prjrvl *NEXT  ;        /* next bi-value                        */
  unsigned char  CNDVAL;        /* driver's condition value             */
  unsigned char  DRVVAL;        /* driven value                         */
  }
prjrvl_list;
```

Cndval, Valeur de la condition : Valeur de la condition de connexion/déconnexion de l'émetteur.

Drvval, Valeur de l'émetteur : Valeur du signal.

Next : Elément suivant de la liste. Sert à obtenir la liste des émetteurs sur le signal.

Glnext, Structure suivante au niveau global : Sert à obtenir la liste de tous les signaux gardés de type BUS.

1.7) Les signaux gardés internes : WRIREG, WRIBUX

Ces signaux gardés sont soit les éléments mémorisants internes de type REGISTER, qui correspondent aux structures **bereg**, soit des signaux BUS de l'interface, utilisés en interne, qui correspondent aux structures **bebux**.

```
typedef struct wrireg                 /* written register             */
  {
  struct wrireg *NEXT     ;           /* next register                */
  struct ptype  *DEPEND   ;           /* dependency list              */
  struct lkdins *LKDINS   ;           /* instance                     */
  struct bereg  *BEREG    ;           /* formal register (model)      */
  unsigned char *CURVAL   ;           /* register’s current value     */
  struct prjrvl *PRJRVL   ;           /* list of projected bi-values  */
  char           ERRFLG   ;           /* error flag                   */
  }
wrireg_list;

typedef struct wribux                 /* written internal bused sig.  */
  {
  struct wribux *NEXT     ;           /* next signal                  */
  struct ptype  *DEPEND   ;           /* dependency list              */
  struct lkdins *LKDINS   ;           /* instance                     */
  struct bebux  *BEBUX    ;           /* formal bused signal (model)  */
  unsigned char *CURVAL   ;           /* register’s current value     */
  struct prjbvl *PRJBVL   ;           /* list of projected bi-values  */
  char           ERRFLG   ;           /* error flag                   */
  }
wribux_list;
```

- **Depend** : Liste des expressions Booléennes dont une des entrées est le signal référencé.

- **Lkdins** : Pointeur vers l'instance qui contient le signal référencé.

- **Bereg, Bebux** : Pointeur vers le signal du modèle correspondant.

- **Curval**, Valeur courante : Pointeur sur la zone du tableau des valeurs courantes, la zone pointée contient la valeur courante du signal.

- **Prjrvl, Prjbvl** : Les bi-valeurs projetées du signal qui représentent les valeurs des émetteurs sur le signal et la condition de connexion/déconnexion de cet émetteur.

- **Errflg,** Drapeau d'erreur : Afin de savoir si antérieurement ce signal a connu une erreur de résolution.

- **Next** : Elément suivant de la liste.

1.8) Tableaux d’association : REDLST

Cette structure permet d'associer la valeur du signal d'une instance en fonction de l'index du signal du modèle correspondant.

```
typedef struct redlst                 /* list of read values          */
  {
  struct redlst *NEXT;                /* next value                   */
```

- **Prjbvl** : Dans le cas d'un signal BUS, pointeur sur la liste des Bi-valeurs projetées.

- **Next** : Elément suivant de la liste

1.5) Signaux internes simples : WRIAUX

```
typedef struct wriaux                  /* written auxiliary signal      */
  {
  struct wriaux *NEXT     ;            /* next auxiliary signal         */
  struct ptype  *DEPEND   ;            /* dependency list               */
  struct lkdins *LKDINS   ;            /* instance                      */
  struct beaux  *BEAUX    ;            /* formal internal sig. (model)  */
  unsigned char *CURVAL   ;            /* aux. signal current value     */
  unsigned char  PRJVAL   ;            /* aux. signal projected value   */
  char           FLAG     ;            /* signal type (delayed)         */
  }
wriaux_list;
```

- **Depend** : Liste des expressions Booléennes dépendantes du signal.

- **Lkdins,** Instance : Pointeur vers l'instance qui contient le signal.

- **Beaux,** Signal du modèle : Pointeur vers le signal du modèle correspondant.

- **Curval,** Valeur courante : Pointeur sur la zone du tableau des valeurs courantes, la zone pointée contient la valeur courante du signal.

- **Prjval,** Valeur projetée : La valeur projetée du signal.

- **Type,** Type du signal : Indique si le signal est retardé. Afin de pouvoir vérifier l'attribut 'STABLE sur le signal en le comparant avec le signal non retardé.

- **Next** : Elément suivant de la liste.

1.6) Les assertions : WRIMSG

Correspond aux structures **bemsg** du modèle.

```
typedef struct wrimsg                  /* assert                */
  {
  struct wrimsg *NEXT     ;            /* next assert           */
  struct bemsg  *BEMSG    ;            /* formal assert (model)  */
  struct lkdins *LKDINS   ;            /* instance              */
  char           ERRFLG   ;            /* assert violation flag  */
  }
wrimsg_list;
```

- **Bemsg**, Clause du modèle : Pointeur vers la clause d'assertion du modèle correspondant.

- **Lkdins** : Pointeur vers l'instance qui contient le signal référencé.

- **Errflg** : Indique si la condition d'assertion est vraie.

- **Next** : Elément suivant de la liste.

```
  char          TYPE     ;            /* signal's type                */
  char          ERRFLG   ;            /* error flag                   */
  }
bussig_list;
```

- **Name** : Il s'agit d'une liste de noms, car ce signal peut posséder plusieurs identificateurs.

- **Depend,** Dépendances : Liste des expressions Booléennes dépendantes du signal.

- **Curval,** Valeur courante : Pointeur sur la zone du tableau des valeurs courantes, la zone pointée contient la valeur courante du signal.

- **Prjval, Prjbvl,** Valeurs projetées : La valeur projetée du signal dans le cas d'un signal simple, une liste de bi-valeurs dans le cas d'un signal gardé, ces bi-valeurs contiennent les valeurs projetées des émetteurs et les valeurs projetées des conditions de connexion/déconnexion.

- **Type,** Type du signal : Dans le cas d'un signal gardé, indique s'il s'agit d'un signal WOR_BIT ou MUX_BIT.

- **Errflg,** Drapeau d'erreur : Dans le cas d'un signal gardé, permet de savoir si antérieurement ce signal avait une erreur de résolution.

- **Next** : Elément suivant de la liste.

1.4) Connecteurs de sortie simples et gardés : WRSSIG, WRBSIG

Ces structures représentent les connecteurs de sortie de l'instance qui vont pointer vers les signaux de sortie des instances (structure **simsig** et **bussig**) également pointés par la structure de simulation (**lkdfig**).

```
typedef struct wrssig                 /* written simple signal        */
  {
  struct wrssig *NEXT     ;           /* next object                  */
  struct beout  *BEOUT    ;           /* formal output port (model)   */
  struct lkdins *LKDINS   ;           /* instance                     */
  struct simsig *SIMSIG   ;           /* actual signal                */
  }
wrssig_list;

typedef struct wrbsig                 /* written bused signal         */
  {
  struct wrbsig *NEXT     ;           /* next signal                  */
  struct lkdins *LKDINS   ;           /* instance                     */
  struct bebus  *BEBUS    ;           /* formal bused signal (model)  */
  struct bussig *BUSSIG   ;           /* actual bused signal          */
  struct prjbvl *PRJBVL   ;           /* list of projected bi-values  */
  }
wrbsig_list;
```

- **Lkdins** : Pointeur vers l'instance qui contient le signal référencé.

- **Bebus, Beout,** Signal du modèle : Pointeur vers le signal du modèle correspondant.

- **Bussig, Simsig,** Signal de sortie des instances : Pointeur vers le signal de sortie des instances.

- **Insname,** Nom d'instance : L'identificateur de la description. Ce nom est unique dans l'espace des noms des instances, cette unicité est garantie par construction.

- **Befig,** Modèle comportemental : Pointeur sur la structure de donnée BEFIG, représentant le modèle comportemental de l'instance.

- **Wrssig,** Connecteurs simples de sortie : Liste des connecteurs de sortie simples de l'instance.

- **Wriaux,** Signaux auxiliaires : Liste des signaux internes simples de l'instance. Correspond aux structures **beaux**.

- **Wribux,** Signaux gardés utilisés en interne : Liste des connecteurs gardés qui sont utilisés en entrée de la description. Correspond aux structures **bebux**.

- **Wribus,** Connecteurs gardés de sortie : Liste des connecteurs gardés de sortie. Correspond aux structures **bebus**.

- **Wrireg,** Signaux internes gardés : Correspond aux structures **bereg**.

- **Wrimsg,** Clauses d'assertion : correspond aux structures **bemsg**.

- **Redlst,** Tableau d'association : Ce tableau d'indirection permet de retrouver la valeur d'un signal instancié, en utilisant l'index du signal correspondant dans le modèle comportemental.

- **Circui** : Ce champ pointe sur le tableau de hash qui sert à la gestions des ROBDD.

- **Next** : Elément suivant de la liste.

1.3) Signaux de sortie simples et gardés des instances : SIMSIG et BUSSIG

Les signaux de sortie d'une instance peuvent être réutilisés en entrée des autres instances. Afin de pouvoir parvenir rapidement à leur valeur, ces signaux sont disponibles directement à partir de la structure de simulation **lkdfig**, chaque instance parvient à ses signaux de sortie par le biais d'une indirection via leurs connecteurs de sortie (structure **wrssig** et **wrbsig**).

```
typedef struct simsig                  /* simple signal               */
  {
  struct simsig *NEXT     ;            /* next simple signal          */
  struct chain  *NAMECHAIN;            /* signal's name               */
  struct ptype  *DEPEND   ;            /* dependency list             */
  unsigned char *CURVAL   ;            /* signal's current value      */
  unsigned char  PRJVAL   ;            /* signal's projected value    */
  }
simsig_list;

typedef struct bussig                  /* bused signal                */
  {
  struct bussig *NEXT     ;            /* next simple signal          */
  struct chain  *NAMECHAIN;            /* signal's name               */
  struct ptype  *DEPEND   ;            /* dependency list             */
  unsigned char *CURVAL   ;            /* signal's current value      */
  struct prjbvl *PRJBVL   ;            /* list of projected bi-values  */
```

Toutes les structures composant la structure de données de simulation ont été conçues pour constituer les éléments d'une liste chaînée.

1.1) Structure de simulation LKDFIG

Cette structure est la racine de la structure de simulation, aussi appelé structure lié (*linked figure*) car elle est le résultat d'une édition de liens entre les structures de données structurelles et les structures de données comportementales.

```
typedef struct lkdfig           /* linked figure                */
  {
  struct lkdfig *NEXT  ;        /* next figure                  */
  char          *NAME  ;        /* figure's name                */
  struct lkdins *LKDINS;        /* list of instances            */
  struct simsig *SIMSIG;        /* list of simple signals       */
  struct bussig *BUSSIG;        /* list of bused signals        */
  }
lkdfig_list;
```

- **Name** : L'identificateur de la description. Ce nom est unique dans l'espace des noms des descriptions, cette unicité est garantie par construction.
- **Lkdins** : Listes des instances des modèles comportementaux.
- **Simsig,** Signaux simples de sortie : Signaux de sortie simples des instances, correspond aux structures **beout**.
- **Bussig,** Signaux gardés de sortie : Signaux gardés de sortie des instances, correspond aux structures **bebus**.
- **Next** : Elément suivant de la liste

1.2) Instance : LKDINS

Cette structure représente chaque instance de modèle. Elle pointe sur le modèle comportemental et par un jeu d'indirection (**REDLST**), associe chaque signal de l'instance avec le signal adéquat du modèle comportemental. Elle permet de représenter les instances des signaux, chaque signal du modèle comportemental de référence trouve donc sa place dans une des sous-structures de la structure d'instance (**lkdins**)

```
typedef struct lkdins           /* linked instance              */
  {
  struct lkdins  *NEXT   ;      /* next instance                */
  char           *INSNAME;      /* instance's name              */
  struct befig   *BEFIG  ;      /* instance model               */
  struct wrssig  *WRSSIG ;      /* written output ports         */
  struct wriaux  *WRIAUX ;      /* written internal signals     */
  struct wribux  *WRIBUX ;      /* written internal bused sig.  */
  struct wrbsig  *WRBSIG ;      /* written bused ports          */
  struct wrireg  *WRIREG ;      /* written registers            */
  struct wrimsg  *WRIMSG ;      /* written asserts              */
  struct redlst  *REDLST ;      /* read values                  */
  struct circuit *CIRCUI ;      /* Circuit pointer used for BDDs */
  }
lkdins_list;
```

LKDFIG	:	Structure racine de la structure de simulation.
LKDINS	:	Instance du modèle pointé par la BEFIG.
BEFIG	:	Racine du modèle comportemental instancié.
BEOUT	:	Signaux de sortie simples du modèle comportemental.
WRSSIG	:	Connecteurs de sortie simple d'une instance.
SIMSIG	:	Signaux de sortie simple de l'instance.
BEBUS	:	Signaux de sortie bussés du modèle comportemental.
BUSSIG	:	Signaux de sortie bussés de l'instance.
WRBSIG	:	Connecteurs de sortie bussés de l'instance.
BEREG	:	Signaux mémorisants internes du modèle comportemental.
WRIREG	:	Signaux mémorisants internes de l'instance.
BEBUX	:	Signaux bussés interne du modèle comportemental.
WRIBUX	:	Signaux bussés interne de l'instance.
BEMSG	:	Les clauses ASSERT du langage VHDL du modèle comportemental.
WRIMSG	:	Les clauses ASSERT de l'instance.
REDLST	:	Liste des associations de signaux (transposition signaux modèle/signaux instance)
DEPEND	:	Liste des dépendances attachées à chaque signal.
PRJBVL	:	Liste des valeurs projetés sur le signal BUS (liste des émetteurs).
PRJRVL	:	Liste des valeurs projetés sur le signal REGISTER (liste des émetteurs).
CURVAL	:	Valeur courante des signaux.
BEAUX	:	Signaux auxiliaires simples du modèle comportemental.
WRIAUX	:	Signaux auxiliaires simples de l'instance.

1)- Structure de données de simulation

Notre simulateur dispose après la compilation de la description du circuit à simuler d'une liste de structures de données structurelles hiérarchiques MBK et d'une liste de structures de données comportementales BEH.

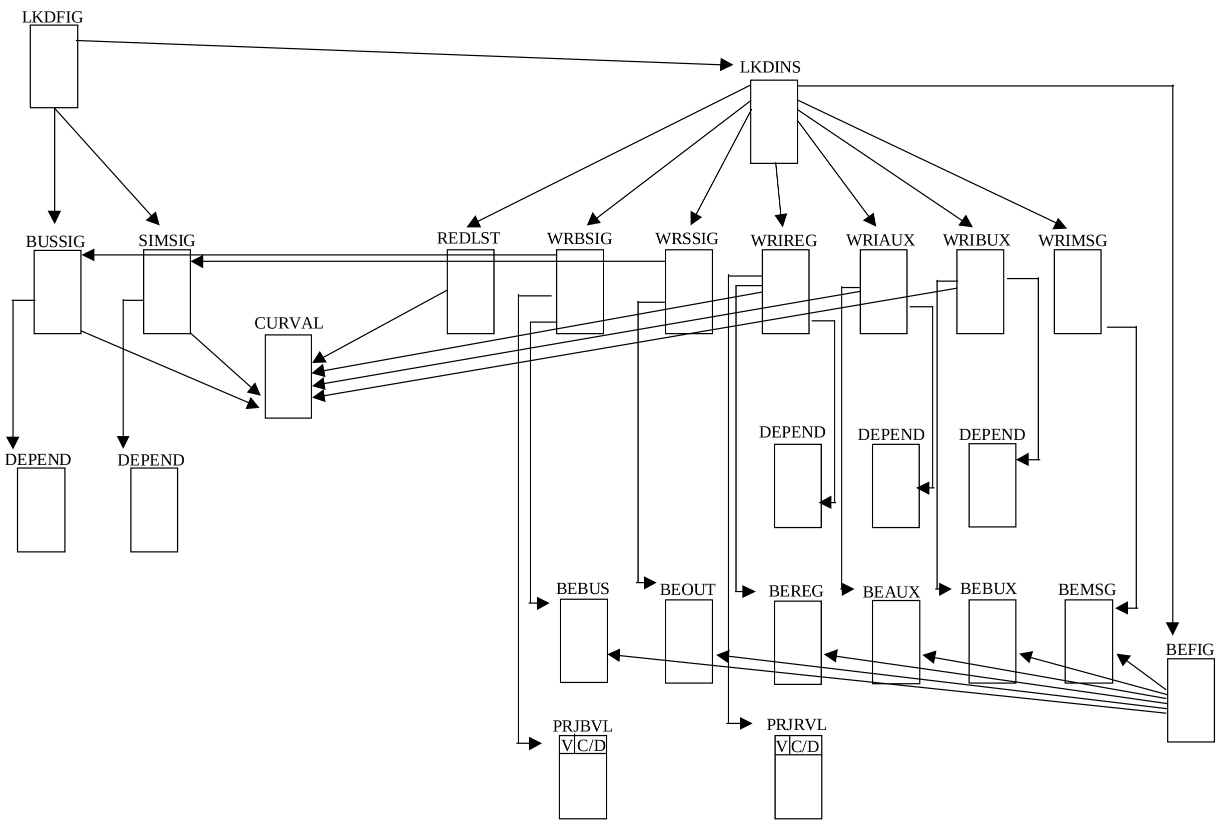
Nous avons vu dans le chapitre 7, que ces structures de données ne sont pas suffisantes afin de pouvoir simuler un circuit.

Nous avons donc défini une structure de données de simulation appelé **LKDFIG**, afin de simuler la description.

Cette structure de données permet de représenter le réseau Booléen représentant la description à simuler. Nous avons factorisé chaque modèle comportemental de la description afin d'optimiser l'occupation en mémoire de la structure.

Ainsi notre structure consiste en un ensemble de modèle comportemental (**befig**), et en un ensemble d'instances de ces modèles (**lkdins**). Un tableau d'indirection (**redlst**) nous permet d'associer chaque signal de l'instance au signal du modèle auquel il est connecté.

Comme expliqué dans le chapitre 7, à chaque signal de l'instance (**wrxxxx**) est attaché sa valeur courante (**curval**) et sa valeur projetée (**prjval**). A l'exception des signaux gardés (BUS ou REGISTER) qui utilise une liste de valeurs projetés (**prjbvl** ou **prjrvl**) représentant les valeurs des émetteurs sur ces signaux ainsi que leur valeur de connexion/déconnexion.



La structure LKDFIG,

Annexe 3

Structures de données de simulation

Signaux internes de type ASSERT : BEMSG

Cette structure de données est utilisée dans BEH afin de représenter les assertions de la description.

L'instruction d'assertion permet de vérifier que certaines conditions sont réalisées lors de la simulation. La clause **severity** permet de définir la gravité du non respect de la condition.

```
typedef struct bemsg            /* assert instruction                 */
  {
  struct bemsg      *NEXT   ;   /* next assertion                     */
  char               LEVEL  ;   /* severity level of the assertion    */
  char              *LABEL  ;   /* instruction's label                */
  char              *MESSAGE;   /* assertion's message                */
  struct chain      *ABL    ;   /* assertion's condition (ABL)        */
  struct node       *NODE   ;   /* assertion's condition (BDD)        */
  }
bemsg_list;
```

- **Label** : L'étiquette de l'assertion.

- **Level,** Niveau de sévérité : Deux niveaux de sévérité sont reconnus : ERROR et WARNING. L'un impose l'arrêt de la simulation, l'autre ne fait qu'afficher le message d'assertion.

- **Message** : Le message à afficher lorsque la condition d'assertion est vraie.

- **ABL** : Pointeur sur un ABL qui représente l'expression logique exprimant la condition de l'assertion.

- **BDD** : Pointeur sur un BDD qui représente l'expression logique exprimant la condition de l'assertion.

- **Next** : Elément suivant de la liste

Cette structure comporte les mêmes champs que BEOUT.

Signaux internes de type BUS : BEBUX

Cette structure de données est utilisée dans BEH afin de représenter des signaux internes de type BUS. Pour un signal vectoriel, chacun des éléments est représenté par une structure, ils sont ordonnés dans l’ordre inverse de la déclaration du vecteur.

Est considéré comme signal interne de type BUS, un connecteur dont la valeur est utilisée par au moins un des signaux internes de la description.

Cette structure comporte les mêmes champs que la structure BEBUS.

Signaux internes de type REGISTER : BEREG

Cette structure de données est utilisée dans BEH afin de représenter les éléments mémorisants internes (REGISTER). Pour un signal vectoriel, chacun des éléments est représenté par une structure, ils sont ordonnés dans l’ordre inverse de la déclaration du vecteur.

Ces éléments mémorisants doivent être assignés à l'intérieur d'une instruction BLOCK et disposent donc d'une condition de garde.

```
typedef struct bereg      /* register                     */
  {
  struct bereg  *NEXT  ;  /* next register                */
  char          *NAME  ;  /* register’s name              */
  struct biabl  *BIABL ;  /* list of drivers (ABL)        */
  struct binode *BINODE;  /* list of drivers (BDD)        */
  }
bereg_list;
```

Elle est constituée de :

- **Name** : L’identificateur du connecteur. Ce nom est unique dans l’espace des noms des signaux, cette unicité est garantie par construction. Pour un signal vectoriel, ce nom est l’identificateur du vecteur suivi d’un espace puis de l’index du signal.

- **BIABL** : Pointeur sur une liste d'émetteurs, chaque émetteur comporte 2 expressions logiques : une représente la valeur émise par l'émetteur et l'autre la condition de connexion de l'émetteur. Chaque expression est représentée par un ABL.

- **BINODE** : Pointeur sur une liste d'émetteurs, chaque émetteur comporte 2 expressions logiques : une représente la valeur émise par l'émetteur et l'autre la condition de connexion de l'émetteur. Chaque expression est représentée par un BDD.

- **Next** : Elément suivant de la liste

- **Node** : Pointeur sur un BDD qui représente l'expression logique rattachée à ce connecteur.

- **Next** : Elément suivant de la liste

Connecteurs de sortie gardé : BEBUS

Cette structure de données est utilisée dans BEH afin de représenter des signaux de sortie de type BUS. Pour un signal vectoriel, chacun des éléments est représenté par une structure, ils sont ordonnés dans l'ordre inverse de la déclaration du vecteur.

Est considéré comme signal de sortie de type BUS un connecteur dont la valeur est définie par au moins un émetteur déconnectable de l'intérieur du circuit.

```
typedef struct bebus             /* bussed output port           */
  {
  struct bebus      *NEXT  ;     /* next bus port                */
  char              *NAME  ;     /* port's name                  */
  struct biabl      *BIABL ;     /* list of drivers (ABL)        */
  struct binode     *BINODE;     /* list of drivers (BDD)        */
  char               TYPE  ;     /* port's type : W or M         */
  }
bebus_list;
```

Elle est constituée de :

- **Name** : L'identificateur du connecteur. Ce nom est unique dans l'espace des noms des signaux, cette unicité est garantie par construction. Pour un signal vectoriel, ce nom est l'identificateur du vecteur suivi d'un espace puis de l'index du signal.

- **Type** : La fonction de résolution attachée aux connecteurs.

- **BIABL** : Pointeur sur une liste d'émetteurs, chaque émetteur comporte 2 expressions logiques : une représente la valeur émise par l'émetteur et l'autre la condition de connexion de l'émetteur. Chaque expression est représentée par un ABL.

- **BINODE** : Pointeur sur une liste d'émetteurs, chaque émetteur comporte 2 expressions logiques : une représente la valeur émise par l'émetteur et l'autre la condition de connexion de l'émetteur. Chaque expression est représentée par un BDD.

- **Next** : Elément suivant de la liste

Signaux internes simples : BEAUX

Cette structure de données est utilisée dans BEH afin de représenter des signaux internes simples. Pour un signal vectoriel, chacun des éléments est représenté par une structure, ils sont ordonnés dans l'ordre inverse de la déclaration du vecteur.

Est considéré comme signal interne simple un signal dont la valeur est guidée par un unique signal à l'intérieur du circuit.

Elle est constituée de :

- **Name** : L'identificateur. Ce nom est unique dans l'espace de noms des signaux, cette unicité est garantie par construction. Pour un signal vectoriel, ce nom est l'identificateur du vecteur suivi d'un espace puis de l'index du signal.

- **Out_ref,** Sorties référencées : Liste des connecteurs de sortie dépendants du signal.

- **Bus_ref**, Sorties référencées : Liste des connecteurs de sortie gardé de type bus dépendants du connecteur d'entrée courant.

- **Aux_ref,** Signaux internes référencées : Liste des signaux internes dépendants du signal.

- **Reg_ref,** Signaux internes référencées : Liste des signaux internes gardés de type register dépendants du signal.

- **Bux_ref,** Signaux internes référencées : Liste des signaux internes gardés de type bus dépendants du signal.

- **Msg_ref** : Liste des instructions d'assertions dépendantes du signal.

- **Dly_ref,** Signaux internes référencées : Liste des signaux retardés dépendants du signal.

- **Next** : Elément suivant de la liste

Connecteurs de sortie : BEOUT

Cette structure de données est utilisée dans BEH afin de représenter des signaux de sortie simple. Pour un signal vectoriel, chacun des éléments est représenté par une structure, ils sont ordonnés dans l'ordre inverse de la déclaration du vecteur. Est considéré comme signal de sortie simple un connecteur dont la valeur est guidée par un unique signal à l'intérieur du circuit.

```
typedef struct beout             /* simple output port          */
  {
  struct beout      *NEXT;       /* next output                 */
  char              *NAME;       /* port's name                 */
  struct chain      *ABL ;       /* port's equation (ABL)       */
  struct node       *NODE;       /* port's equation (BDD)       */
  }
beout_list;
```

Elle est constituée de :

- **Name** : L'identificateur du connecteur. Ce nom est unique dans l'espace des noms des signaux, cette unicité est garantie par construction. Pour un signal vectoriel, ce nom est l'identificateur du vecteur suivi d'un espace puis de l'index du signal.

- **ABL** : Pointeur sur un ABL qui représente l'expression logique rattachée à ce connecteur.

- **Next,** Connecteur suivant : Elément suivant de la liste.

Données génériques : BEGEN

Cette structure de données est utilisée dans BEH afin de représenter des constantes globales relatives au circuit. Il s'agit d'une liste chaînée, ordonnée dans l'ordre inverse de la déclaration faite dans le fichier de description. Pour un signal vectoriel, chacun des éléments est représenté par une structure, là aussi, ils sont ordonnés dans l'ordre inverse de la déclaration du vecteur.

```
typedef struct begen            /* generic                     */
  {
  struct begen *NEXT ;          /* next generic                */
  char         *NAME ;          /* generic's name              */
  char         *TYPE ;          /* generic type's name         */
  void         *VALUE;          /* generic's value             */
  }
begen_list;
```

Elle est constituée de :

- **Name** : L'identificateur de l'objet. Ce nom est unique dans l'espace de noms des signaux, cette unicité est garantie par construction.

- **Type** : Le type de la constante, limité au type *natural*.

- **Value** : La valeur de la constante.

- **Next**, Donnée suivante : Elément suivant de la liste.

Signaux internes et Connecteurs d'entrée : BERIN

Cette structure de données est utilisée dans BEH afin de représenter des signaux d'entrée. Pour un signal vectoriel, chacun des éléments est représenté par une structure, ils sont ordonnés dans l'ordre inverse de la déclaration du vecteur. Est considéré comme signal d'entrée soit un connecteur d'interface utilisé comme entrée, soit un registre interne.

Un connecteur d'interface est considéré comme entrée, quand il peut piloter des signaux à l'intérieur du circuit.

```
typedef struct berin            /* primary inputs                      */
  {
  struct berin *NEXT;           /* next input                          */
  char         *NAME;           /* input's name                        */
  struct chain *OUT_REF;        /* dependent simple output             */
  struct chain *BUS_REF;        /* dependent bussed output             */
  struct chain *AUX_REF;        /* dependent simple internal signal    */
  struct chain *REG_REF;        /* dependent internal register         */
  struct chain *BUX_REF;        /* dependent bussed internal signal    */
  struct chain *MSG_REF;        /* dependent assertion                 */
  struct chain *DLY_REF;        /* dependent delayed internal signal   */
  }
berin_list;
```

- **Bemsg,** liste des instructions d'assertion.

- **Errflg,** Indicateur d'erreur : S'il est mis à zéro, cela signifie que la figure construite est cohérente et complète.

- **Type,** Type de la description : BDD ou ABL.

- **Circui** : Ce champ pointe sur le tableau de hash qui sert à la gestions des ROBDD.

- **User,** Champ utilisateur : Cela permet aux différents utilisateurs de la structure BEH de l'enrichir avec des informations supplémentaires.

- **Next,** Figure suivante : Elément suivant de la liste.

Connecteurs : BEPOR

Cette structure de données est utilisée dans BEH afin de représenter un connecteur un-bit. Il s'agit d'une liste chaînée ordonnée dans l'ordre inverse des connecteurs déclarés dans le fichier de description.

Pour un tableau (*array*), chacun des éléments du vecteur sera représenté par une structure de données. Ils sont également ordonnés dans l'ordre inverse de la déclaration du signal vectoriel.

```
typedef struct bepor      /* port                              */
  {
  struct bepor *NEXT     ;/* next port                         */
  char         *NAME     ;/* port's name                       */
  char          DIRECTION;/* port's mode (I, O, B, Z, T)       */
  char          TYPE     ;/* port's type (B, M or W)           */
  }
bepor_list;
```

Une structure connecteur est composée des champs suivants :

- **Name** : L'identificateur du connecteur dans la figure. Ce nom est unique dans l'espace de noms des signaux, cette unicité est garantie par construction. Pour un signal vectoriel, ce nom est l'identificateur du vecteur suivi d'un espace puis de l'index du signal.

- **Direction** : Indique le mode du connecteur.
  IN.
  OUT.
  INOUT.
  BUSED OUTPUT : Le connecteur de sortie, est gardé de type BUS.
  BUSED INOUT : Le connecteur d'entrée-sortie est de type BUS.

- **Type** : Indique le type de connecteur :
  BIT.
  MUX_BIT.
  WOR_BIT.

Les descriptions qui suivent définissent un objet en fonction des éléments qui le composent. Toutes les structures ont été conçues pour constituer les éléments d'une liste chaînée.

Circuit **BEFIG**

représente la description comportementale d'un circuit.

```
typedef struct befig            /* behaviour figure                     */
  {
  struct befig      *NEXT  ;     /* next figure                          */
  char              *NAME  ;     /* figure's name                        */
  struct bereg      *BEREG ;     /* list of internal signals (REGISTER) */
  struct bemsg      *BEMSG ;     /* list of assert instructions         */
  struct berin      *BERIN ;     /* list of inputs (registers and ports)*/
  struct beout      *BEOUT ;     /* list of output ports                */
  struct bebus      *BEBUS ;     /* list of output ports (BUS)          */
  struct beaux      *BEAUX ;     /* list of internal signals            */
  struct beaux      *BEDLY ;     /* internal signals used for 'DELAYED  */
  struct bebux      *BEBUX ;     /* list of internal signals (BUS)      */
  struct bepor      *BEPOR ;     /* list of ports in reverse order      */
  struct begen      *BEGEN ;     /* list of generic data                */
  struct circuit    *CIRCUI;     /* Pointer on Circuit for BDDs         */
  struct ptype      *USER  ;     /* reserved for user's applications    */
  char               ERRFLG;     /* error flag                          */
  char               TYPE  ;     /* description type                    */
  char               FLAG  ;     /* description flag                    */
  }
befig_list;
```

Elle est constituée de :

- **Name** : C'est l'identificateur de la figure, il 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.

- **Bepor**, liste des connecteurs : Ils définissent son interface logique, ils sont ordonnés dans l'ordre inverse des déclarations contenues dans le fichier. Ce sont des connecteurs de l'interface.

- **Begen**, liste des données génériques : Ce sont les données figurant à l'intérieur d'une clause GENERIC.

- **Beout,** liste des connecteurs en sortie : Ce sont les connecteurs de sortie simples (par opposition aux connecteurs de type bus).

- **Bebus,** liste des connecteurs en sortie de type bus.

- **Berin,** liste des signaux internes et des connecteurs d'entrée : Ce sont tous les signaux qui peuvent apparaître dans une expression logique.

- **Beaux,** liste des signaux internes simples.

- **Bedly,** la liste des signaux retardés : ce sont des signaux créés implicitement lorsque l'on rencontre l'attribut **STABLE**.

- **Bereg,** liste des registres.

La diversité des formats de représentation de l'informations est un des aspects les moins satisfaisants pour le développeur d'application, ces formats sont souvent liés à un environnement de conception spécifique. La chaîne de CAO Alliance propose une solution à ce besoin d'uniformisation par le biais d'une plate-forme de développement pour outils de CAO. Cette plate-forme repose sur une structure de données orientée objet destinée à fournir un modèle non ambigu des différentes vues d'un circuit : vues physiques, structurelles et comportementales.

Les objectifs de ces structures de données sont les suivants :

- rendre les différents outils de CAO - générateurs, extracteurs, preuve formelle ... - indépendants des formats de fichiers: il y a un découpage complet entre les outils qui ne manipulent que des objets en mémoire et les fonctions de lecture et d'écriture des fichiers sur disque. Seules ces fonctions sont dépendantes du format de fichier.

- fournir aux outils de CAO des objets non ambigus à sémantique fixe, garantissant un certain niveau de cohérence de façon à faciliter la communication entre les outils.

Ces objectifs tendent à garantir et à faciliter le développement d'outils de CAO communiquant par construction. Ils simplifient considérablement l'interface avec le monde extérieur.

Ces structures, fondamentalement, ne sont pas conçus pour être des structures de travail. Elles sont conçus afin de représenter des données. Les utilisateurs de ces structures de données doivent donc les enrichir pour pouvoir les employer comme structure de travail.

La structure de donnée représentant la vue structurelle (interconnexion de composant), était déjà définie [PET94]. Nous avons spécifié la structure de données permettant de représenter une description comportementale au sens défini en annexe 1.

La vue comportementale : Structure BEH

Une description comportementale comporte plusieurs parties :

Son entité (*entity*) contient son nom, son interface et ses données génériques. Les signaux de son interface peuvent être séparés en trois catégories : Signaux d'entrée simples, signaux de sortie simples, signaux d'entrée de type BUS et enfin signaux de sortie de type BUS.

Sa clause d'architecture contient les signaux internes de la description (signaux simples et registres) ainsi que ses assignations ou ses assertions.

Nous allons détailler les objets de la structure de données appelée Beh, qui permet de représenter une description comportementale VHDL. Les objets nécessaires à la représentation de la vue comportementale sont au nombre de douze. A chaque objet correspond une structure et une seule, celle ci possède des champs dont la sémantique est prédéfini et détaillée dans les paragraphes qui suivent.

Annexe 2

Structures de données comportementale

La vue comportementale : Structure BEH

Circuit **BEFIG**
Connecteurs : BEPOR
Données génériques : BEGEN
Signaux internes et Connecteurs d’entrée : BERIN
Connecteurs de sortie : BEOUT
Connecteurs de sortie gardé : BEBUS
Signaux internes simples : BEAUX
Signaux internes de type BUS : BEBUX
Signaux internes de type REGISTER : BEREG
Signaux internes de type ASSERT : BEMSG

Non supporté

8.12 - Return Statement
Non supporté

8.13 - Null Statement
Non supporté

9 - CONCURRENT STATEMENTS
9.1 - Block Statement
Les Blocks gardés sont supportés
Les ports et les generics ne sont pas supportés dans les clauses **block**

9.2 - Process Statement
Non supportés

9.3 - Concurrent Procedure Call
Non supportés

9.4 - Concurrent Assertion Statement
supportés

9.5 - Concurrent Signal Assignment Statement
L'option **Guard** est supportée.
L'option **Transport** est non supportée.
Les informations temporelles sont non supportés.

9.6 - Component Instantiation Statement
Supportés pour les **port map**

9.7 - Generate Statement
Non supporté

7 - EXPRESSIONS

7.2 – Operators

Les opérateurs logiques sont supportés.
Le seul opérateur d'addition supporté est **&**
La multiplication et la division ne sont pas supportés
Le reste, le modulo, la valeur absolue, et les opérateurs de puissance ne sont pas supportés.

7.3 - Operands

Les littéraux sont supportés
Les agrégats ne sont pas supportés
Les appels de fonctions exceptés les fonctions prédéfinies ne sont pas supportés.
Les expressions qualifiés ne sont pas supportés.
Les conversions de type ne sont pas supportés.
Les allocateurs ne sont pas supportés

7. 4 - Static Expressions

Les expressions statiques ne sont pas supportés.

7.5 - Universal Expressions

Les expressions universelles ne sont pas supportés.

8 - SEQUENTIAL STATEMENTS

8.1 - Wait Statement

Non supporté.

8.2 - Assertion Statement

Non supporté

8.3 - Signal Assignment Statement

Non supporté

8.4 - Variable Assignment Statement

Non supporté

8.5 - Procedure Call Statement

Non supporté

8.6 - If Statement

Non supporté

8.7 - Case Statement

Non supporté

8.9 - Loop Statement

Non supporté

8.10 - Next Statement

Non supporté

8.11 - Exit Statement

4.3.1 - Object Declarations
Les déclarations de signaux sont supportés.
Les déclarations de constantes sont supportés.

4.3.2 - File Declarations
Non supportés.

4.3.3 - Interface Declarations
Les déclarations d'interface sont supportés.
Le mode **Buffer** n'est pas autorisé.
Le mode **Linkage** est supporté aux interfaces des **Entity**. Les **Linkages** sont interprétés comme des signaux de mode inconnu.

4.3.4 - Alias Declarations
Non supportés.

4.4 - Attribute Declarations
Non supportés.
Toutefois, un attribut prédéfini de signal est supporté : **stable**

4. 5 - Component Declarations
Supportés.
Les clauses **Port** sont supportés.

5 - SPECIFICATIONS
5.1 - Attribute Specifications
Non supportés

5.2 - Configuration Specifications
Non Supportés

5.3 - Disconnection Specifications
Non supportés

6 - NAMES
6.1 - Simple Names
Supportés.

6.2 - Selected Names
Non Supportés

6.3 - Indexed Names
Supportés, en rappelant la restriction que seuls les vecteurs à une dimension sont autorisés.

6.4 - Sliced Names
Supportés, en rappelant la restriction que seuls les vecteurs à une dimension sont autorisés.

6.5 - Attribute Names
Limités dans le compilateur a l'attribut de signal prédéfini : **stable.**

Différents styles de description sont supportés : Structurel et Flot de données.

1.3 - Configuration Declarations
Non supportés.

2 - SUBPROGRAMS AND PACKAGES
2.1 - Subprogram Declarations
Les procédures et les fonctions ne sont pas supportés.

2.2 - Subprogram Bodies
Non supportés

2.3 - Subprogram Overloading
Non supportés.

2.4 - Resolution Functions
Les fonctions de résolution définies par l'utilisateur ne sont pas supportés.
Quelques fonctions de résolution prédéfinies sont supportées afin de traiter les signaux multiplexeurs distribués et les signaux préchargés.

2.5 - Package Declarations
Non supportés

2.6 - Package Bodies
Non supportés.

3 - TYPES
3.1 - Scalar Types
Les types prédéfinis d'énumération (**bit**, et **severity_level**) sont supportés.
Les types prédéfinis entiers sont non supportés.
Les types prédéfinis physiques sont non supportés.
Les types prédéfinis à virgule flottantes sont non supportés.

3.2 - Composite Types
Le type prédéfini vectorisé (**bit_vector**) est supporté
Seulement les vecteurs à une dimension sont supportés.

3.3 - Access Types
Non supportés

3.4 - File Types
Non supportés

4 - DECLARATIONS
4.1 - Type Declarations
Les déclarations de type ne sont pas supportés.

4.2 - Subtype Declarations
Non supportés.

Ces instructions regroupent donc :

Les instructions block. Dans notre sous-ensemble, toutes les instructions block doivent posséder une condition de garde.

Toutes les assignations concurrentes (simple, sélective et conditionnelle), à condition qu'elles soient à délai nul et qu'elles concernent les types supportés par notre sous-ensemble (à savoir **bit**, **bit_vector**, **mux_bit**, **mux_vector, wor_bit**, **wor_vector**, **reg_bit** et **reg_vector**).

Les instructions d'assertions concurrentes.

Cas des signaux gardés :

Un signal gardé **bus** ou **register** est associé à travers son type à une fonction de résolution. Une fonction de résolution détermine à un instant donné, la valeur d'un signal selon l'état de ses émetteurs.

Dans notre sous-ensemble VHDL, toutes les fonctions de résolution sont prédéfinis et il n'est pas possible d'en définir de nouvelles. Il existe 3 types pour les signaux gardés :

- **reg_bit** et **reg_vector** associés au mot clé **register** pour modéliser un élément mémorisant.
- **wor_bit** et **wor_vector** associés au mot clé **bus** pour modéliser un bus **OU** câblé distribué.
- **mux_bit** et **mux_vector** associés au mot clé **bus** pour modéliser un bus multiplexeur distribué.

1.3) Autres éléments de VHDL non conservés

Les directives de VHDL relatives à la gestion de contexte (paquetage et configuration) et la gestion de bibliothèque sont exclues du sous-ensemble. D'une manière générale, l'ensemble des types prédéfinis a été simplement étendu pour permettre la modélisation d'éléments séquentiels et de bus.

1.4) LRM

Nous présentons ici le sous-ensemble du langage VHDL, à l'aide du Manuel de Référence du Langage. (IEEE – Std 1076 – 1987), en utilisant les mêmes numéros de chapitre et de section.

1 - DESIGN ENTITIES AND CONFIGURATIONS

1.1 - Entity Declarations

Les clauses **Generic** sont supportés dans une description comportementale.
Les clauses **Port** sont supportés.

1.2 - Architecture Bodies

Plusieurs corps d'Architecture peuvent être spécifiés pour un **entity**.

signal a **: bit;**
signal b**,** c**,** d **: bit_vector (**0 **to** 3**);**
signal x **: mux_vector (**0 **to** 3**) bus;**

Trois informations sont attachées à chaque déclaration.
- Le nom ou l'identificateur du signal.
- Son type.
- Sa taille pour un tableau.
- Si le signal est un signal « gardé ». Cette information est fournie par la présence du mot clé **bus**.

La déclaration des composants internes permet de connaître la liste des modèles utilisés.

component and2
port (....**);**
end component;

Cette déclaration ressemble à la définition d'une vue externe.

La partie descriptive d'une description structurelle représente l'interconnexion des instances des composants déclarés dans la partie déclarative. La connexion de chaque instance est décrite par trois informations :
- Le nom de l'instance.
- Le nom du modèle.
- La connexion des connecteurs de l'instance et un signal.

Notre sous-ensemble admet les types de connexions implicites et explicites. Il est possible de ne pas connecter certains connecteurs de sortie.

1.2.2) La forme comportementale

Une autre manière de décrire la vue interne d'un circuit est la description comportementale. On tente de modéliser le circuit par un ensemble d'instructions concurrentes.

Dans la partie déclarative d'une description comportementale, ne peuvent apparaître que des déclarations de signaux et de constantes.

Les constantes sont de type **bit** ou **bit_vector** et doivent être initialisées. Les signaux peuvent être de type **bit** ou **bit_vector**, ou des types résolus **mux_bit**, **mux_vector**, **wor_bit** ou **wor_vector**. Les registres doivent être explicitement déclarés par un signal gardé de type **reg_bit** ou **reg_vector** accompagné du mot-clé **register**.

Dans la partie descriptive, notre sous-ensemble comportemental autorise toutes les instructions concurrentes à l'exception des :
- Process
- Instructions d'instanciations (cf Description Structurelle)

Remarque : **linkage** signifie que le mode du signal est inconnu, il est surtout utilisé lorsque le fichier VHDL est obtenu par traduction d'un fichier écrit dans un autre langage.

1.2) La vue interne

La description de la vue interne peut se faire sous deux formes :
- Forme structurelle
- Forme comportementale

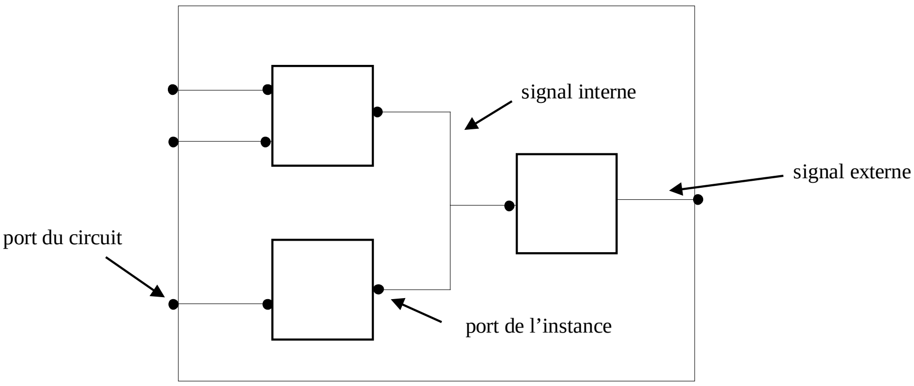
1.2.1) La forme structurelle

Dans VHDL, l'aspect structurel et l'aspect comportemental ne sont pas dissociés. En effet, une description peut être à la fois structurelle et comportementale. Dans le sous-ensemble que nous définissons, ce mélange est interdit, cette restriction ne représente pas une limitation très forte pour les concepteurs. De plus, elle permet l'interopérabilité de notre sous-ensemble VHDL avec d'autres formats tels que Spice, EDIF, …

La description d'une vue interne, qu'elle soit structurelle ou comportementale se fait par la clause **architecture**, suivie d'une partie déclarative et d'une partie descriptive.

Une description structurelle est une description dans laquelle le circuit est décrit en terme des composants qui la constituent.

Dans ce type de description, il faut donc, d'une part, spécifier les composants utilisés dans le circuit, et, d'autre part, la manière dont ces composants sont reliés.



Deux types de signaux sont utilisés pour relier les composants. Les signaux dits externes, sont reliés à un et un seul connecteur du circuit. Les signaux dits internes ne relient que les connecteurs des instances. Les signaux internes portent un nom quelconque, tandis que les noms externes portent le nom du connecteur du circuit.

Dans une description structurelle, la partie déclarative regroupe :
- La déclaration des signaux internes.
- La déclaration des composants utilisés.

La déclaration des signaux internes se fait par la clause **signal**.

1) Définition du sous-ensemble VHDL

Deux règles vont orienter les définitions du sous-ensemble VHDL :

- L'utilisation de description synchrone.
- Simplification de tous les aspects non indispensables.

Qu'appelons nous une description synchrone ?
Dans une description synchrone, on s'intéresse aux circuits numériques rythmés par une horloge. Ce qui signifie :
- Qu'il n'y a pas de boucle combinatoire instable.
- Qu'on exclut toute représentation de délai physique attaché à une affectation. Autrement dit, toutes les affectations se font à 0 sec. de délai.

Dans la méthodologie de conception dans laquelle s'intègre Asimut, la tâche de vérification temporelle incombe à un outil spécialisé. Cette méthode a été largement utilisée dans l'industrie et s'est concrétisée dans la conception de circuits de grande complexité. Aussi, nous avons restreint Asimut aux descriptions VHDL synchrones.

En VHDL, la description d'un circuit revêt deux aspects distincts :
- La vision externe du circuit.
- La description de l'intérieur du circuit.

1.1) La vue externe

La description de la vue externe se fait en utilisant le mot clé **entity**,
entity amd2901 **is**
generic(……)**;**
port(......**);**
end amd2901**;**

Dans notre sous-ensemble, la clause **generic** permet de préciser des données constantes génériques au gré de l'utilisateur sur le circuit. Nous trouverons donc des déclarations de constantes décrites de la façon suivante :
- Le mot clé **constant**
- Le nom ou l'identificateur de la constante.
- Le type de la constante, nous n'acceptons que le type **natural**.
- La valeur de la constante.

La clause **port** permet de définir les connecteurs du circuit. A chaque connecteur (ou port) sont attachés quatre informations :
- Le nom ou l'identificateur du connecteur.
- Son type. (Les seuls types utilisés sont **bit**, **bit_vector** et les types résolus **mux_bit**, **mux_vector**, **wor_bit** et **wor_vector**).
- Son mode : **in**, **out**, **inout** et **linkage**.
- Sa contrainte, si c'est un tableau.
- Si le connecteur est un connecteur « gardé ». Cette information est fournie par la présence du mot clé **bus**.

Annexe 1

Le sous-ensemble VHDL

d'abstraction fonctionnelle et de preuve Booléenne. Il exclut les instructions Process et l'utilisation des délais non nuls dans les assignations.

La méthode de simulation que nous proposons a pu être mise en œuvre dans un prototype appelé Asimut. Ce prototype a été comparé à deux simulateurs VHDL (Cadence et Synopsys). Nous avons prouvé que cette méthode était viable et que les performances de notre prototype le rendait apte aux développement de circuits complexes. En effet, Asimut est très performant lors de la simulation de net-lists extraites où les signaux sont déclarés par des bits individuels. Cadence et Synopsys le surpassent lorsqu'ils peuvent utiliser la vectorisation des données.

Le sous-ensemble du langage VHDL que nous avons défini a été utilisé avec succès, lors du développement de circuits de recherche de notre laboratoire d'accueil. Il est également utilisé par l'ensemble des outils de la chaîne de CAO Alliance.

Lors de la conception de notre prototype, nous avons été en mesure d'adopter une architecture modulaire. Cette architecture permet de le considérer comme une plate-forme de développement utilisant l'algorithme event-driven. Cette plate-forme a été mise en œuvre dans plusieurs autres logiciels de notre laboratoire d'accueil.

1) Perspectives

Notre technique de représentation d'une description VHDL par une structure de données contenue en mémoire, n'est pas limitée à la simulation logique. Il est possible de développer différentes fonctionnalités utilisant les mêmes structures.

Ainsi, la simulation logico-temporelle peut être mise en œuvre en modifiant notre structure de simulation. Il faut développer un échéancier temporel et rajouter les informations de délais dans nos structures de données. Un prototype utilisant ces solutions a été développé dans notre laboratoire d'accueil.

De même, notre simulateur peut utiliser des techniques de simulation compilées et interprétées. Ainsi, la structure de simulation accepte actuellement des instances de modèle VHDL, elle peut de la même manière accepter des instances de fonction C (cas d'une technique compilée) ou des instances d'interpréteur (cas d'une technique interprétée). Un prototype de simulateur mixte (description VHDL)/(Description langage C) a également été développé dans notre laboratoire d'accueil.

Asimut utilise un noyau de simulation découplé de la structure de données, ce noyau peut donc être modifié afin de traiter d'autres valeurs que les états logiques d'un circuit. Ainsi, en utilisant notre noyau de simulation et en modifiant ses fonctions d'évaluations et de mise à jour, il est possible de mettre en œuvre différents outils logiciels utilisant la propagation d'événements. Des outils ont été développés dans notre laboratoire d'accueil, utilisant cette méthode, tels qu'un simulateur de fautes ou un outil de calcul de consommation.

2) Conclusion

Nous nous sommes proposés d'explorer une nouvelle technique de simulation logique pour des circuits intégrés numériques synchrones de grande complexité et de vérifier la pertinence de cette technique en développant un prototype.

Tous les logiciels de simulation développés jusqu'à présent utilisent, soit une technique interprétée, soit une technique compilée. Ces techniques ont le défaut d'être soit lente (cas de la technique interprétée), soit difficile à mettre en œuvre (cas de la technique compilée).

Nous avons cherché à obtenir un compromis viable entre le temps de compilation et le temps d'exécution : conserver un temps de compilation raisonnable pour accélérer la mise au point de descriptions comportementales, obtenir un temps d'exécution faible pour une vérification efficace des net-lists extraites.

Nous avons pu atteindre ce résultat par l'utilisation d'une nouvelle méthode de simulation fondée sur l'évaluation directe des expressions Booléennes représentées en mémoire par différents types de graphes : les arbres binaires (ABL) et les graphes de décision binaire (BDD).

L'efficacité de notre méthode de simulation est également due à la restriction du langage de description VHDL à un sous-ensemble. Ce sous-ensemble est utilisable par des outils de manipulation de fonctions Booléennes tels que les outils de synthèse logique,

Chapitre IX

Perspectives et conclusion

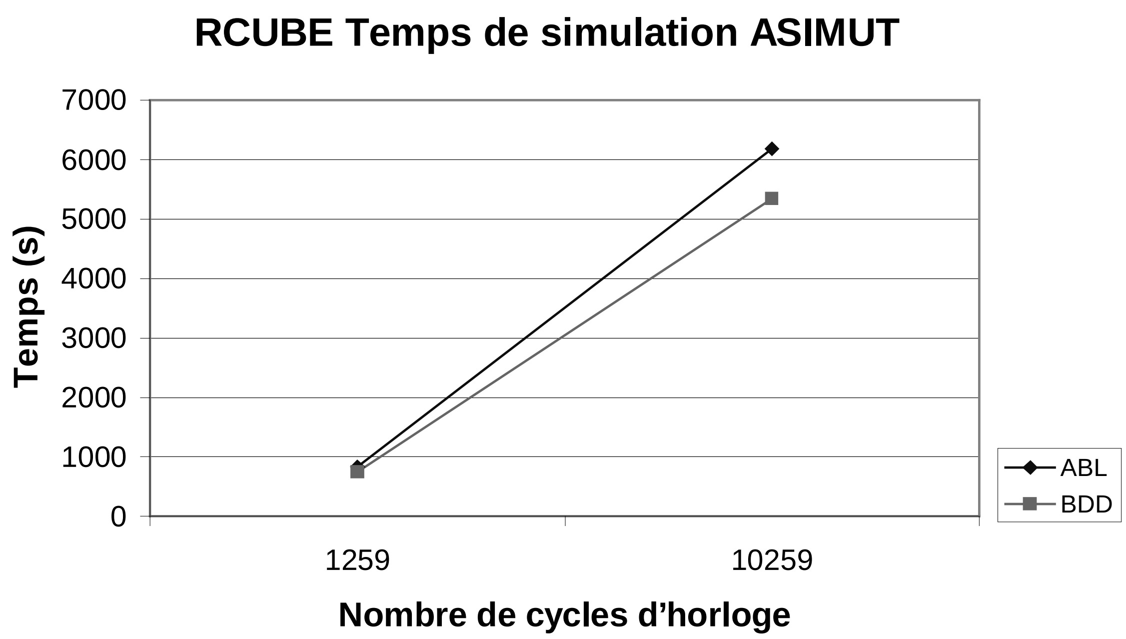
3) Conclusion

Tout au long du processus de conception d'un circuit intégré, la simulation logique est utilisée à des fins de vérification dans divers contextes. A une extrémité de ce processus se trouve la validation des spécifications. A l'autre la vérification de la net-list extraite du dessin des masques.

Lors de la validation des spécifications, la description du circuit est généralement comportementale. A cette phase, le temps de compilation est un facteur important pour optimiser le cycle simulation - analyse – mise au point. A l'autre extrémité, la vérification de la conformité de la net-list extraite, c'est le temps d'exécution qui devient prépondérant.

Dans cette phase, la simulation de circuits numériques utilisant une technique basée sur l'évaluation de graphes peut être relativement performante face aux simulateurs compilés commerciaux. Cette performance se manifeste lors de la simulation de descriptions structurelles ou comportementales extraites, où les signaux sont déclarés comme des bits individuels.

En revanche, lors de la validation des spécifications où la description contient des signaux vectorisés, les simulateurs tirent profit de la vectorisation intrinsèque des processeurs sur lesquels ils s'exécutent (les processeurs 32 bits peuvent exécuter une opération sur 32 bits en une instruction), pour accélérer la phase d'exécution. En contrepartie, la phase de compilation où l'outil cherche à optimiser le code généré devient plus lente.



2.3) Analyse

2.3.1) Temps

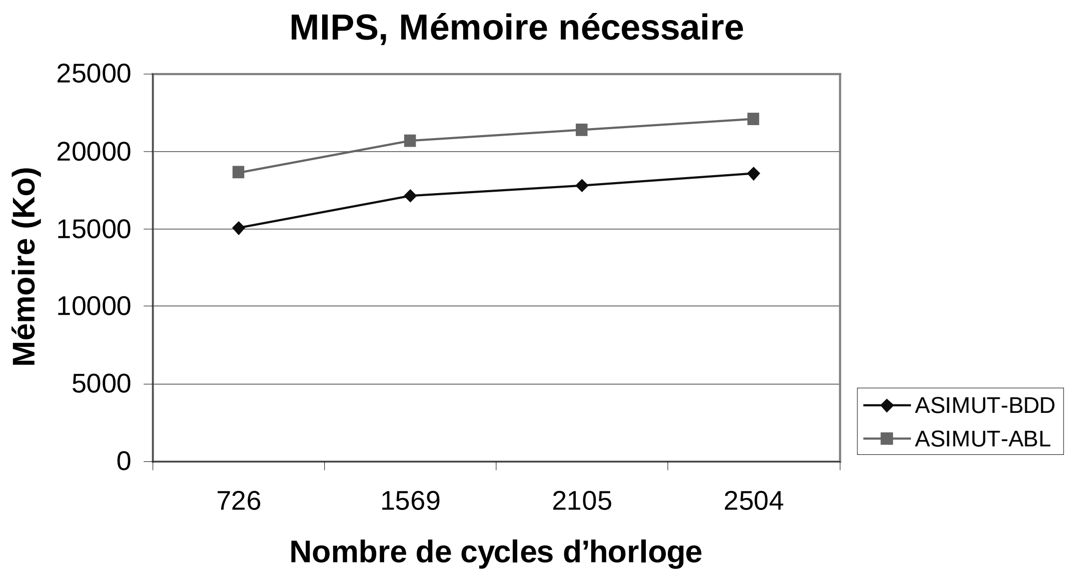
Lors de l'utilisation de BDD, les temps de compilation sont étroitement dépendants de l'ordonnancement des variables dans le fichier de descriptions,. En effet, il n'y a pas de réordonnancement lors de la phase de construction des BDDs. Ainsi les circuits MIPS et Rcube ne sont pas compilables lorsque l'ordonnancement des variables est mauvais. Nous avons ordonnancés ces circuits manuellement afin de réduire au maximum la taille des graphes à construire.

L'utilisation des ABLs permet de s'affranchir de ces problèmes d'ordonnancement, la compilation d'un circuit en utilisant des ABLs est ainsi plus rapide qu'en utilisant des BDDs. Ainsi par rapport aux meilleurs temps de compilation avec des BDDs, l'utilisation des ABLs permet de réduire d'au moins 50% les temps de compilation.

Les temps d'exécution montrent les BDDs sont plus rapides (5% à 50%) que les ABLs. En effet, ces derniers, imposent des appels à des fonctions spécifiques d'évaluation alors que les BDDs ne nécessitent qu'un parcours de graphe.

2.3.2) Ressources matérielles

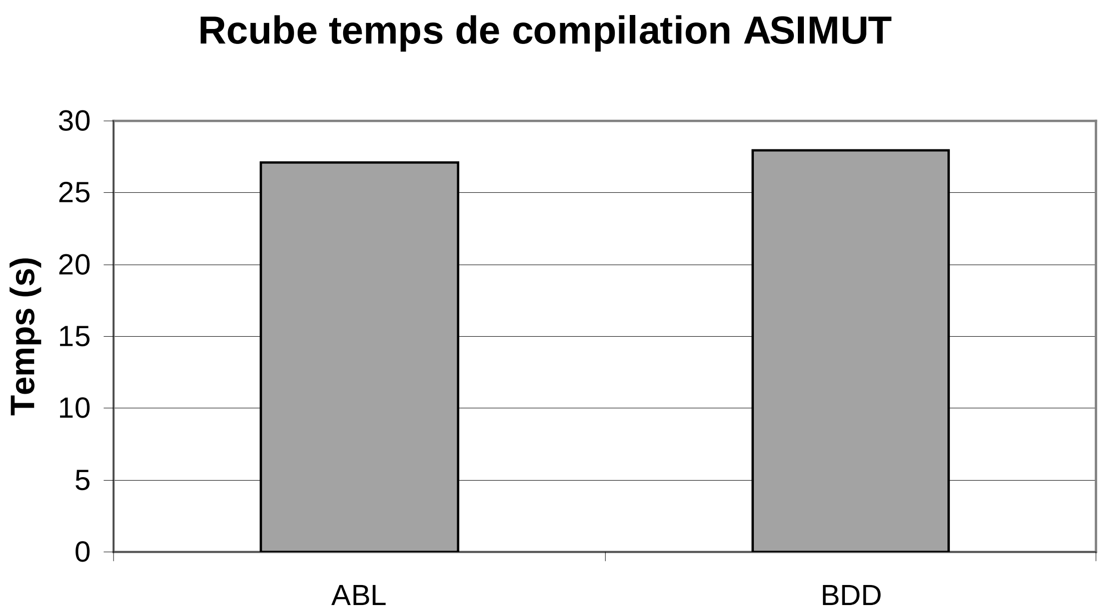
La mémoire nécessaire lors de l'utilisation des BDDs est du même ordre de grandeur que lors de l'utilisation des ABLs, lorsque le fichier est ordonnancé. Cette mémoire peut croître jusqu'à dépasser les capacités des ordinateurs lorsque des BDDs sont utilisés avec un fichier ordonnancé de manière non optimale.



Nous observons que les BDDs sont plus longs à construire que les ABLs, mais ils sont moins consommateurs de mémoire et plus rapide à exécuter.

Le circuit simulé ici est utilisé dans l'enseignement de la micro-électronique, la description du Mips a été optimisée de façon à permettre aux étudiants de pouvoir simuler ce circuit avec des BDDs. En effet, avec un ordonnancement non optimal des variables, le Mips dans certains cas n'est pas compilable, les graphes à construire deviennent trop important pour être contenu en mémoire.

Circuit Rcube



vectorisés, la vitesse de simulation, pour une description vectorisée, est bien meilleure pour Cadence et Synopsys qui deviennent plus rapide qu'Asimut.

2)BDD/ABL

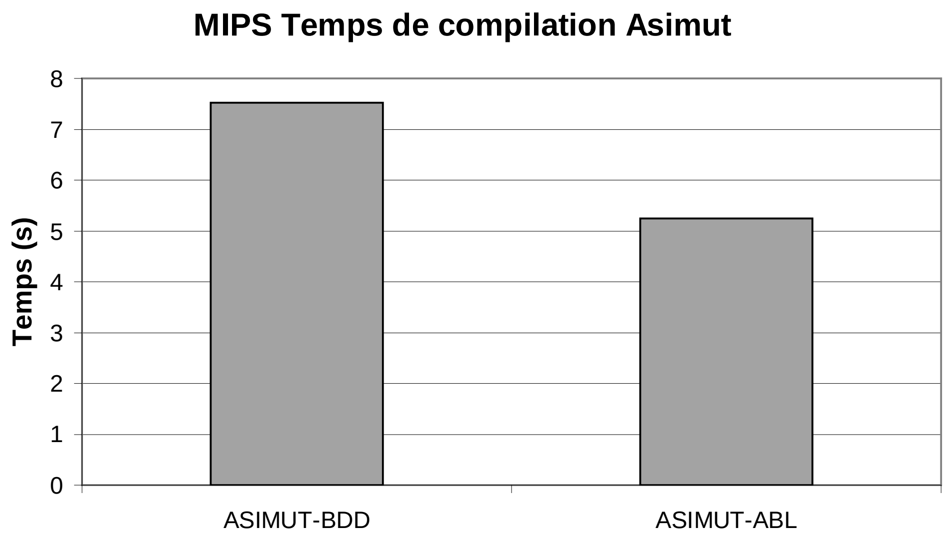
2.1) Protocole de mesure.

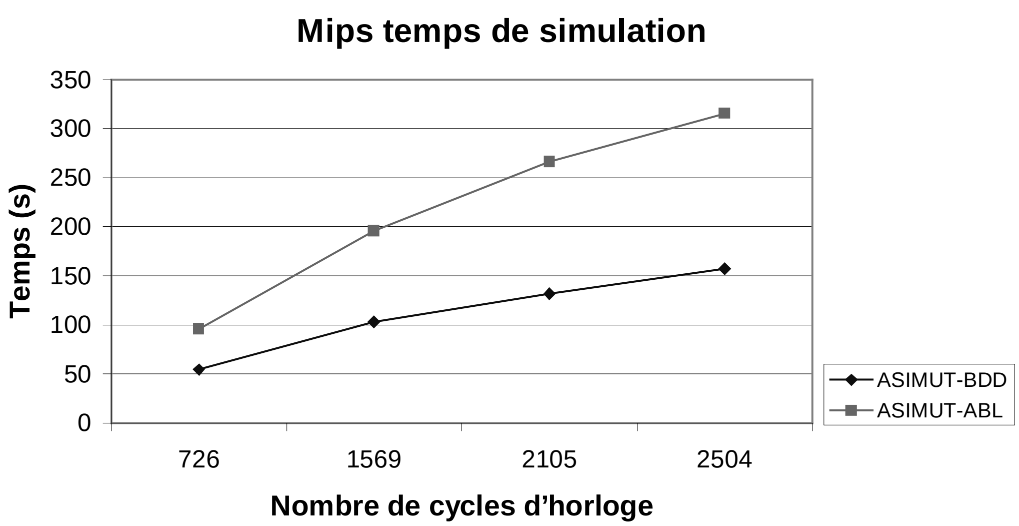
Les circuits suivants ont été simulés avec Asimut, en utilisant des BDDs et des ABLs. Les mesures sont réalisés sur la même station de travail : une station Sun Sparc 10, Processeur SuperSparc 50 (70 SPECint 92 ~ 70 Mips), 416 Mo de RAM, 400 Mo de swap sous SunOS 4.1.3.

Le programme GNU time a été utilisé sur plusieurs simulations consécutives, les graphes ci-dessous ont été réalisés avec la moyenne des résultats.

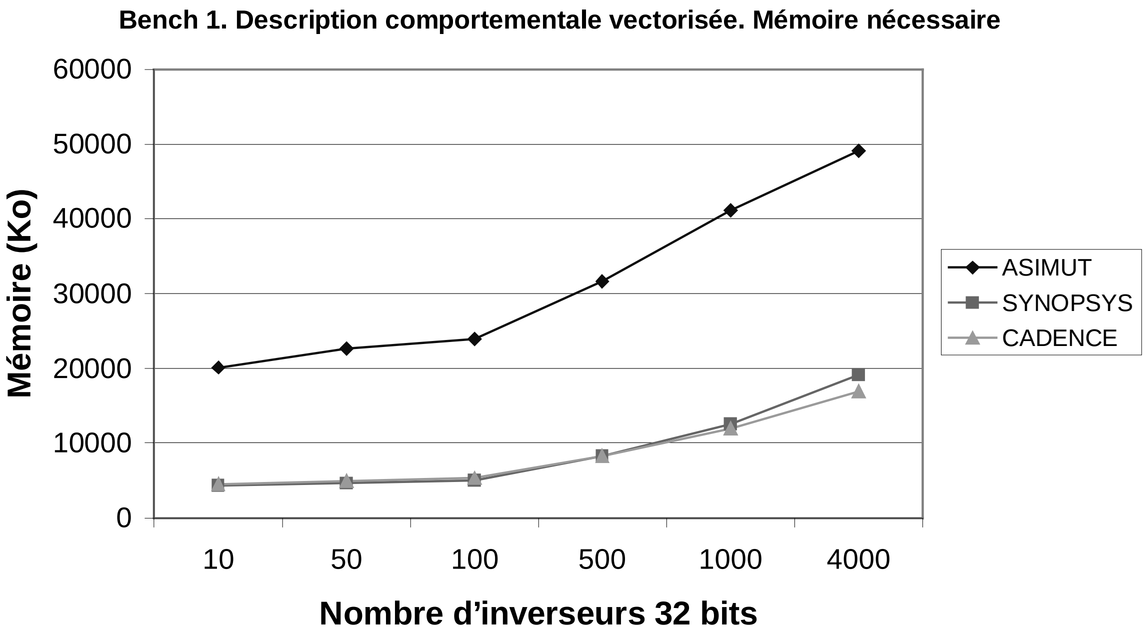
2.2) Comparaison

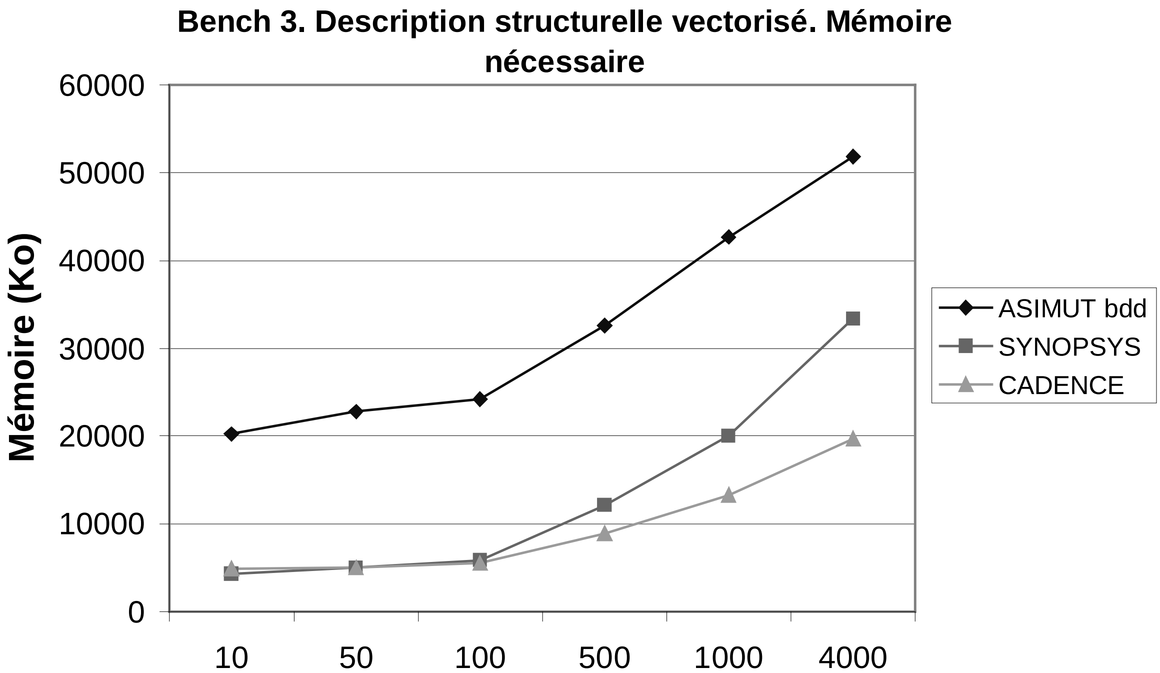
Circuit MIPS





1.4.2) Ressources nécessaires





La seule différence entre les Benchs 1 et 3 et les deux derniers benchs, est la vectorisation des signaux.

Cette dernière étude montre que Synopsys et Cadence exploite effectivement la vectorisation des signaux à l'intérieur des descriptions. Comparé aux benchs 1 et 3 non

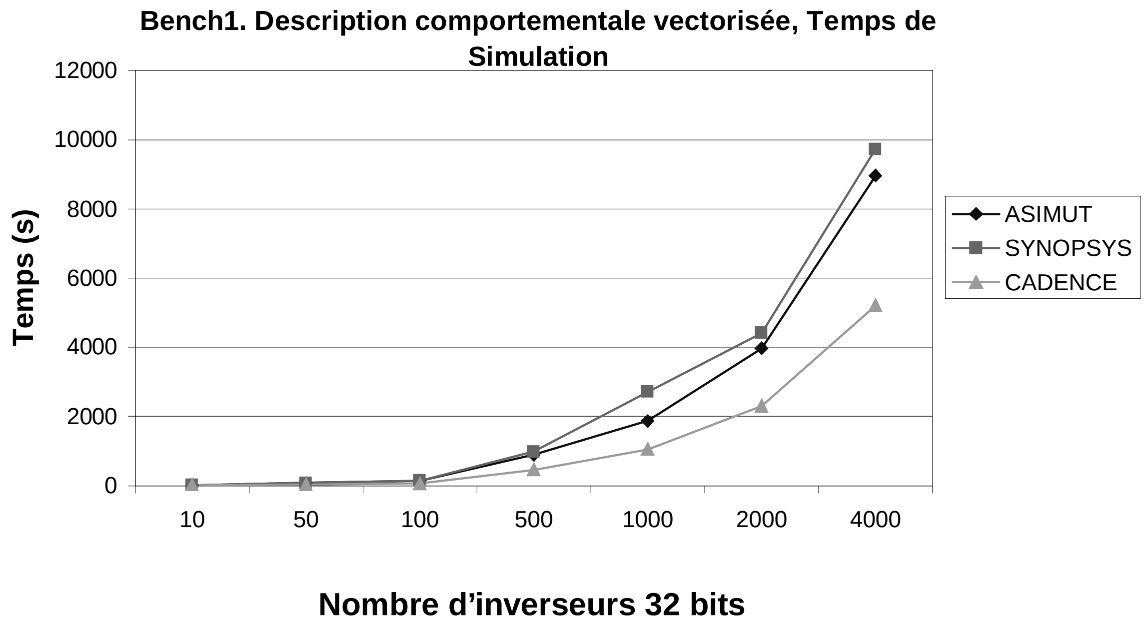
Afin de vérifier cette hypothèse, nous avons imaginé une autre série d'essais basés sur les benchmarks 1 et 2 en utilisant des signaux vectorisés.

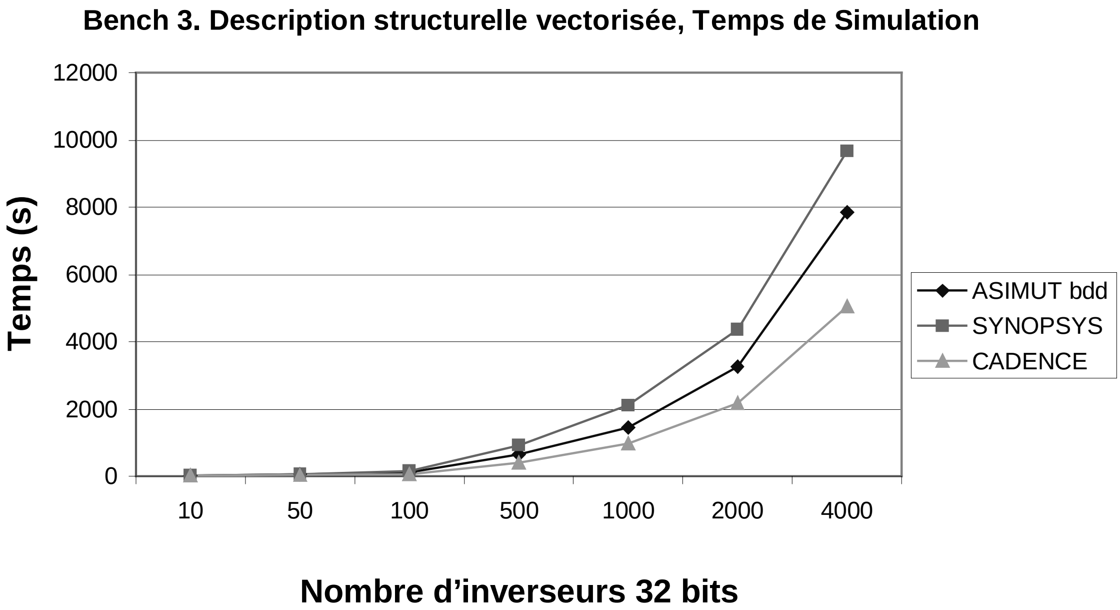
1.4) Benchmarks 1 et 3 avec des vecteurs.

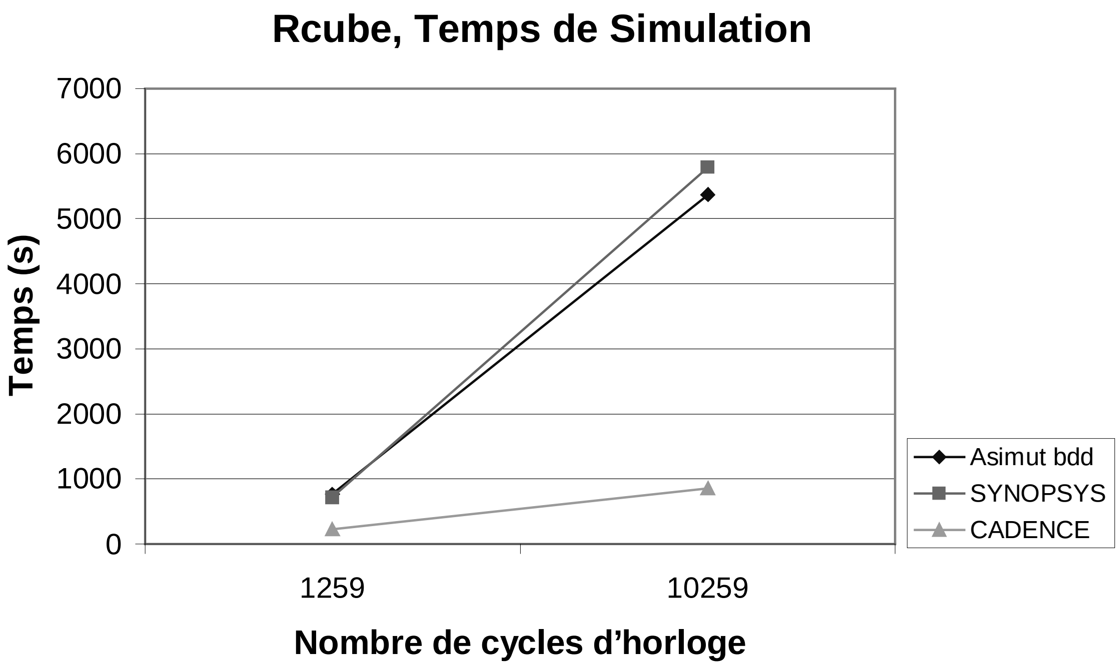
Le Bench 1 devient une description comportementale de n inverseurs décrits en flot de données du type : S_{i+1} (31 downto 0) <= NOT S_i (31 downto 0). Nous obtenons ainsi 32 tranches identiques, chaque tranche équivaut au circuit du Bench 1.

Le Bench 2 devient une description structurelle de n cellules inverseuses décrites en flot de données comme dans le Bench 1 vectoriel.

1.4.1) Temps de simulation







1.3.2) Ressources matérielles nécessaires

La mémoire nécessaire n'a pas pu être mesuré lors de la simulation de Rcube, car la quantité de mémoire nécessaire entrainait l'utilisation de swap et de changement de contexte qui perturbait notre outil de mesure.

Analyse :

Sur les exemples de circuits "réels" exposés ci-dessus. On remarque que Cadence est le plus rapide des simulateurs. De plus, sa consommation mémoire est très optimisée puisqu'on ne constate pas d'augmentation de la mémoire entre 3776 et 30776 patterns pour RCube.

Ces exemples de circuits réels contredisent les résultats observés sur la chaîne d'inverseurs. Asimut est celui qui consomme le plus de mémoire. Cette situation est conforme à notre attente puisque la technique de simulation d'Asimut est justement basée sur la représentation du circuit sous forme de structures de données en mémoire.

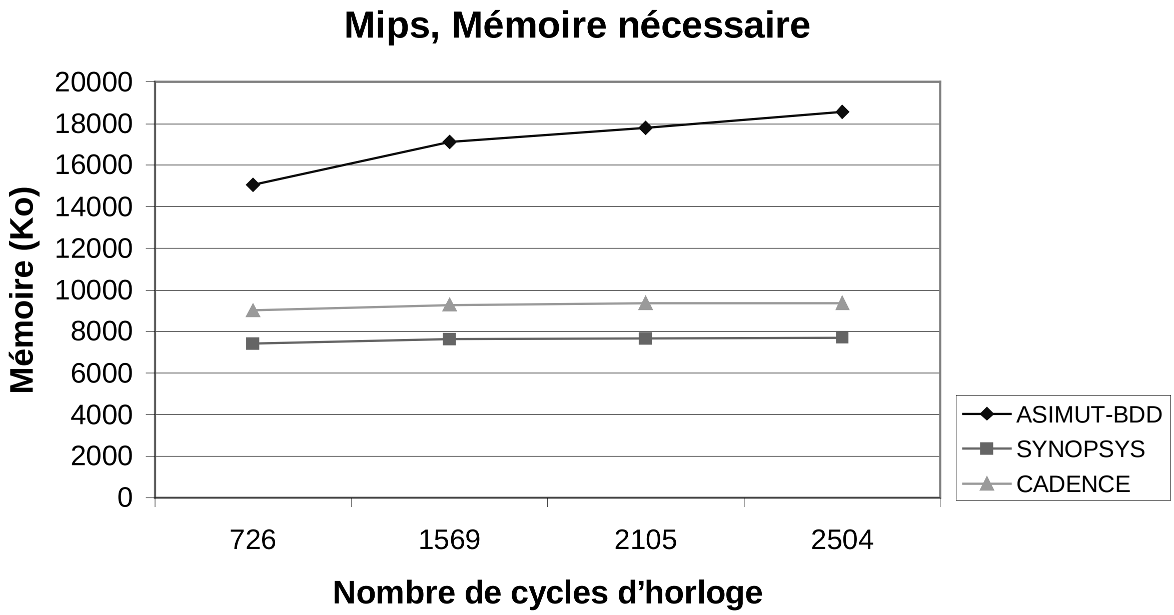
Par contre concernant le temps d'exécution, le rapport entre Synopsys et Asimut est inversé (sur le Mips, Synopsys est plus de trois fois plus rapide). Nous pensons que cela provient de l'exploitation de la vectorisation des signaux à l'intérieur des simulateurs compilés. Ainsi quand on rencontre une assignation du type :

```
a(31 downto 0) <= b(31 downto 0) or d(31 downto 0)
```

Asimut représente cette assignation à l'aide de 32 Bdds (1 Bdd pour chaque bit de a).

Il semble que grâce à la longueur de mot des processeurs sur lesquels ils s'exécutent (le processeur Sparc des stations de travail est un processeur 32 bits), Synopsys et Cadence sont capables de traduire cette assignation en un code compilé utilisant une fois l'instruction OR du processeur. Ceci leur permet de consommer moins de mémoire, de plus une évaluation suffit afin de calculer la valeur du signal a, alors qu'Asimut nécessite 32 évaluations.

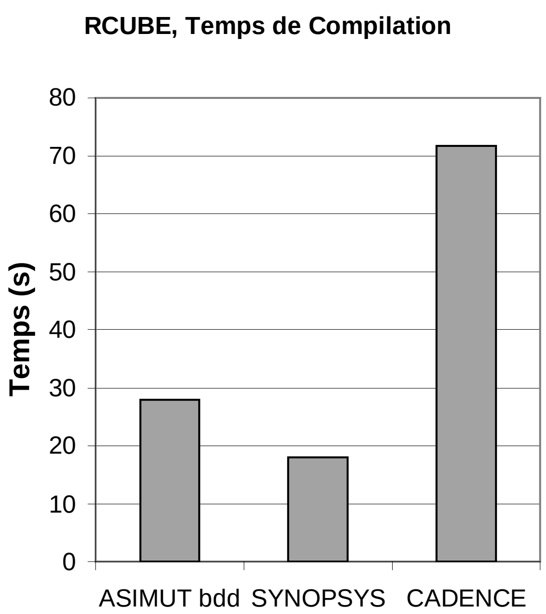
1.2.2) Ressources matérielles



1.3) Rcube

Il s'agit d'un routeur de 8 liens série 1Gbit/s.
Nous simulons ici la description comportementale du circuit avec deux séries de stimuli.
La circuit après synthèse possède une complexité d'environ 100.000 portes.

1.3.1) Temps de simulation



Synopsys et Cadence présente la même courbe de consommation mémoire, mais passé un certain seuil (25 Mo pour le bench 1, 45 Mo pour le bench 2), Cadence utilise un gestionnaire mémoire plus optimisé qui lui permet de ralentir l'augmentation de sa consommation mémoire.

Paradoxalement, nous pensions que le fait de représenter en mémoire les expressions Booléennes de la description à l'aide de graphes allait faire d'Asimut un simulateur nécessitant plus de mémoire que les simulateurs compilés. Nous constatons que sur ce cas d'école, où les expressions Booléennes sont très nombreuses et très petites, Asimut est le simulateur qui nécessite le moins de mémoire.

Nous allons poursuivre notre étude sur des circuits réels développés dans notre laboratoire d'accueil.

1.2) MIPS

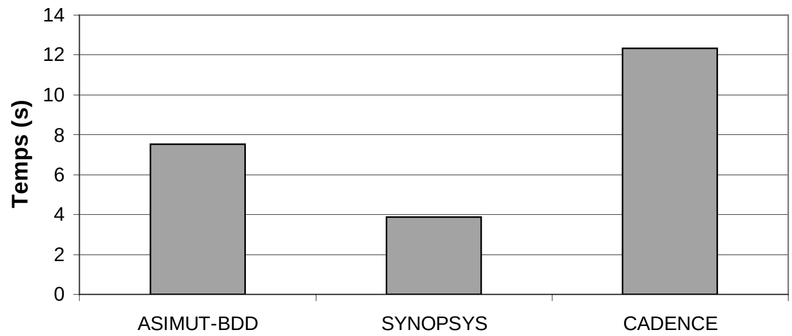
Le Mips R3000 est un processeur RISC 32 bits pipe-line.

Nous simulons la description purement comportementale (assignations concurrentes) de ce circuit. Le circuit après synthèse possède une complexité d'environ 20.000 portes.

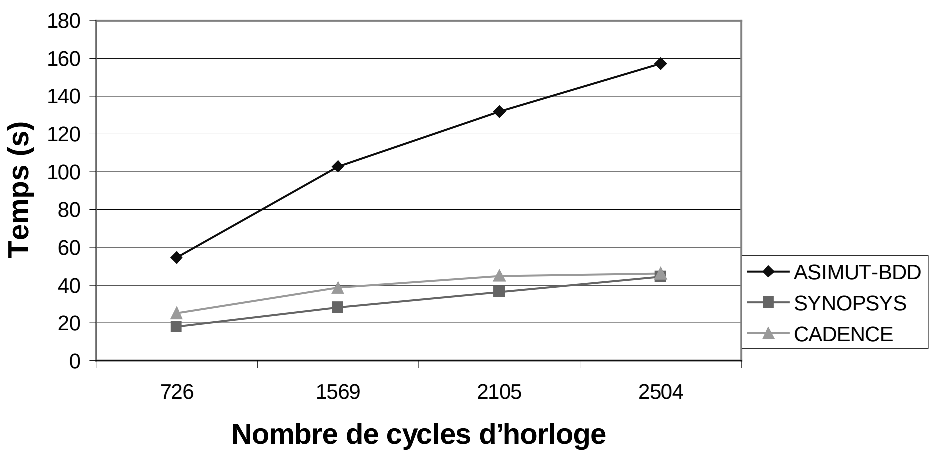
1.2.1) Temps de simulation

Nous avons simulé ce processeur avec différents ensembles de stimuli.

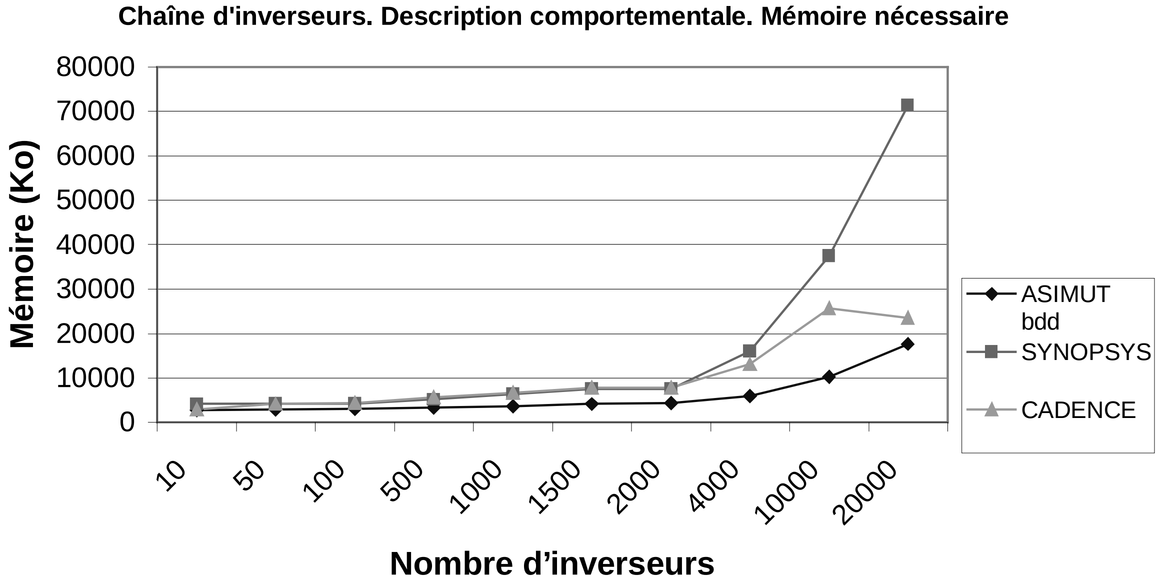
Mips, temps de compilation

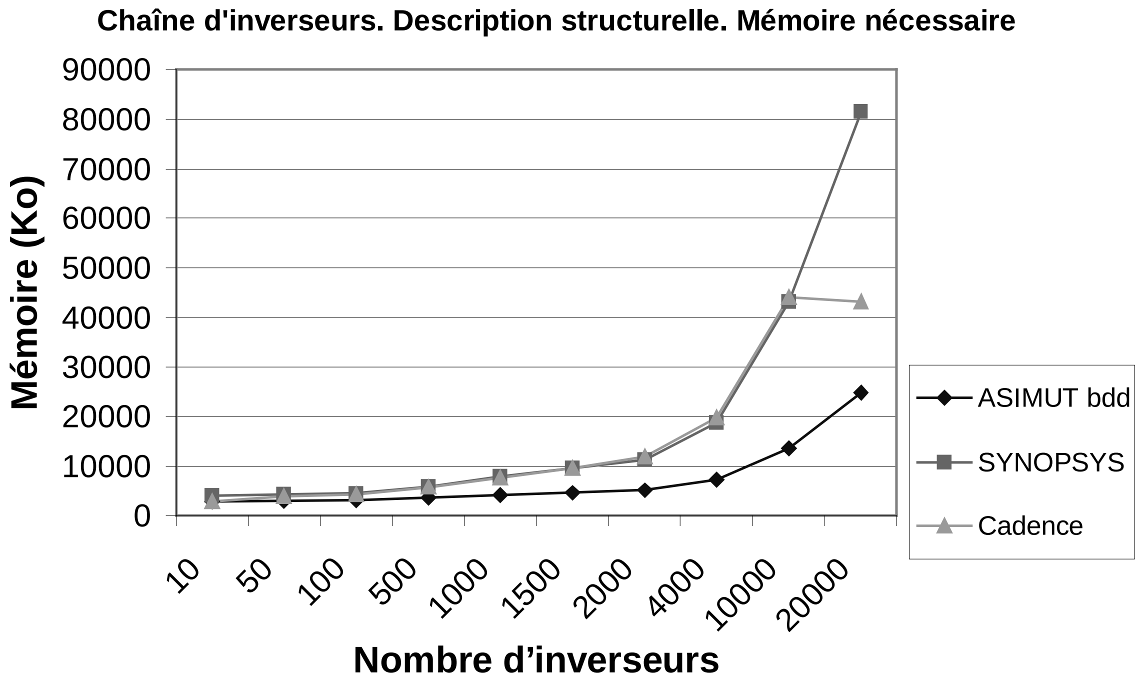


Mips, Temps de Simulation



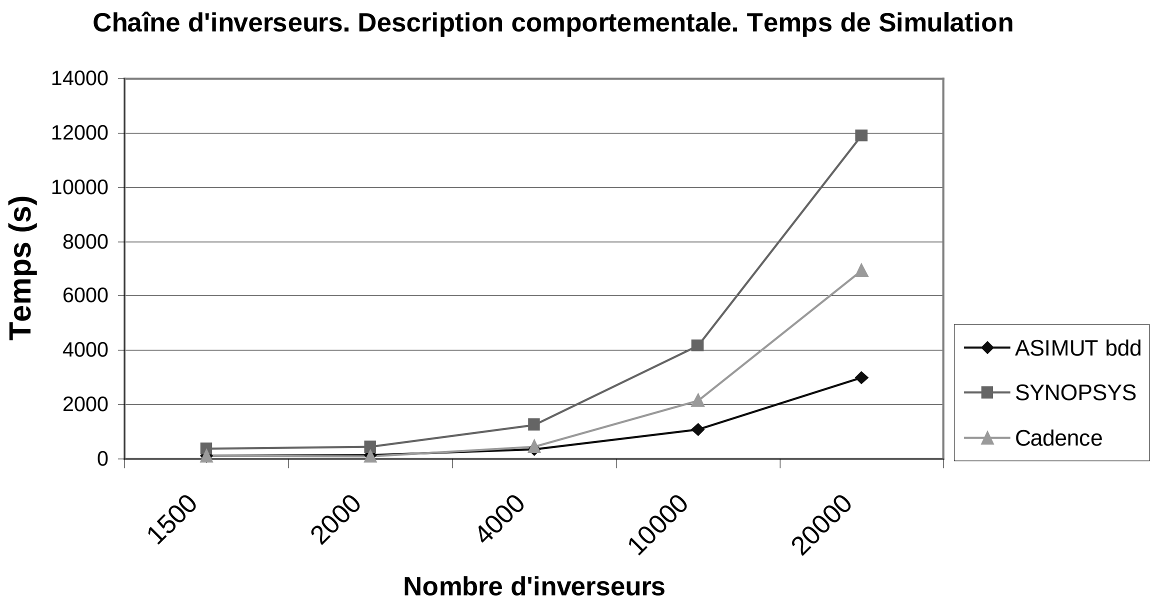
1.1.2) Ressources matérielles

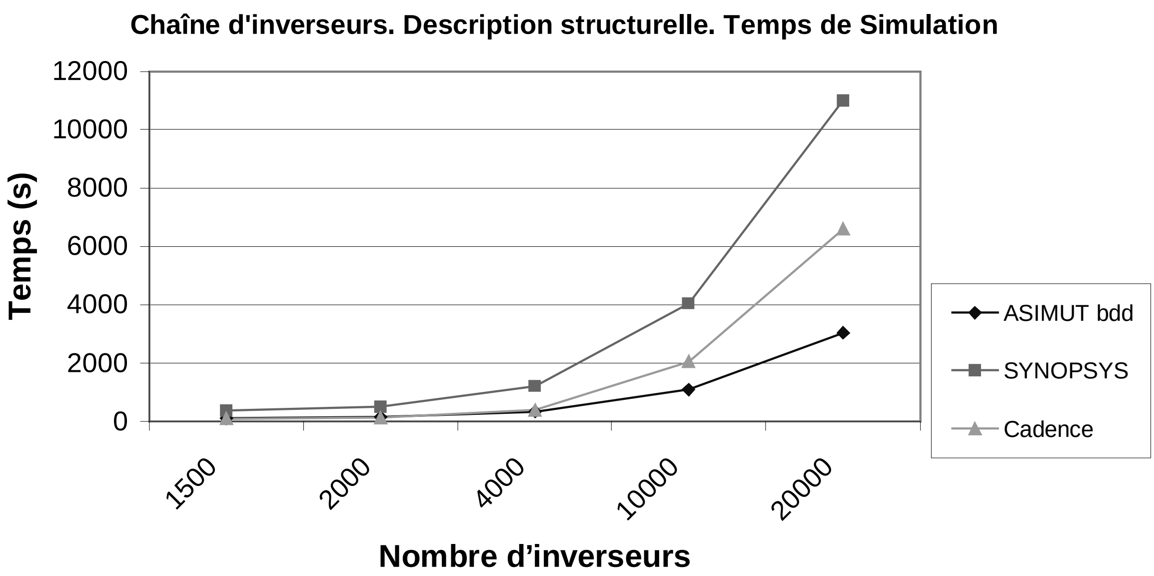




Il faut noter que lors de l'étude de M. HUEBER, publié en 1991, et menée sur une station Sun 3/110, deux simulateurs avaient échoués lors du BENCH2. Le temps de compilation de certains simulateurs dépassait 36 heures au delà de 1500 instances.

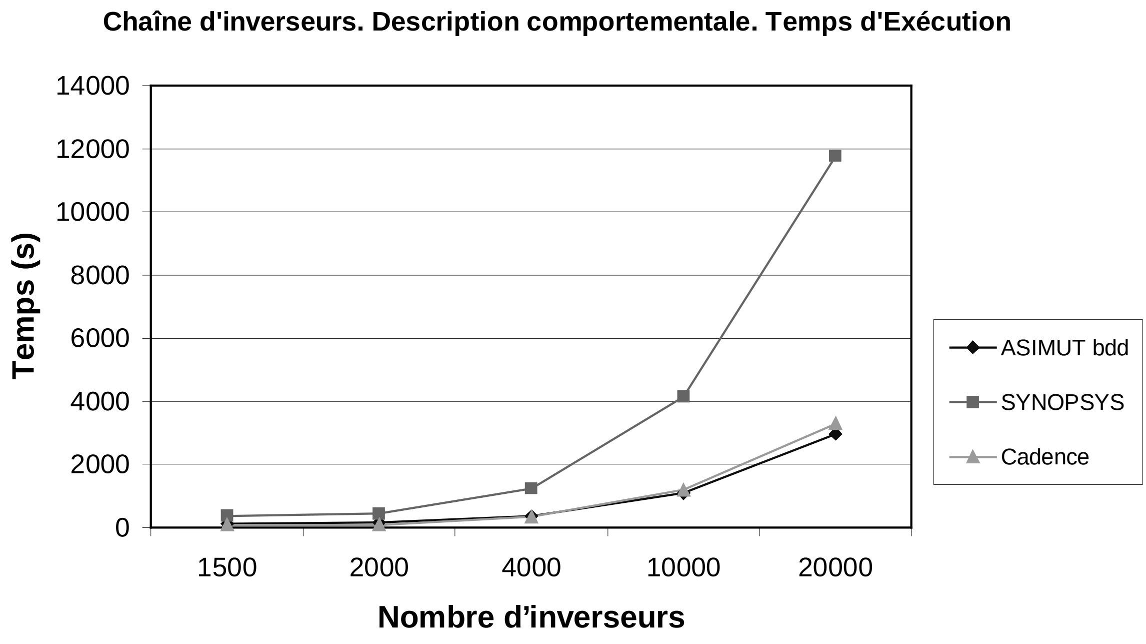
Comme, on peut le constater sur les graphiques ci-dessus, nous sommes loin de ces résultats. Tous les simulateurs parviennent sans problèmes à compiler et simuler 20000 instances.

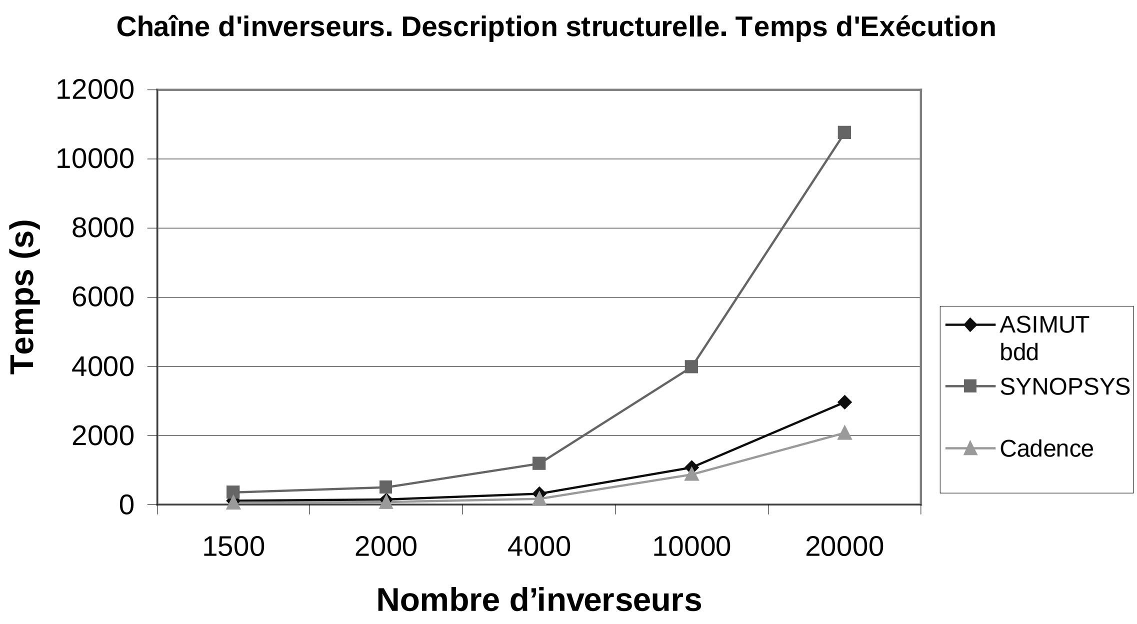




En résumé, Synopsys sur les quatre benchmarks réalisés est jusqu'à 40% plus lent que Cadence, qui lui même est deux fois plus lent qu'Asimut. En effet, Synopsys compile rapidement (même ordre de grandeur qu'Asimut), mais exécute le plus lentement des trois simulateurs. Cadence a fait le choix d'une optimisation poussée lors de sa phase de compilation, mais il obtient en contrepartie une très bonne vitesse d'exécution (même ordre de grandeur qu'Asimut).

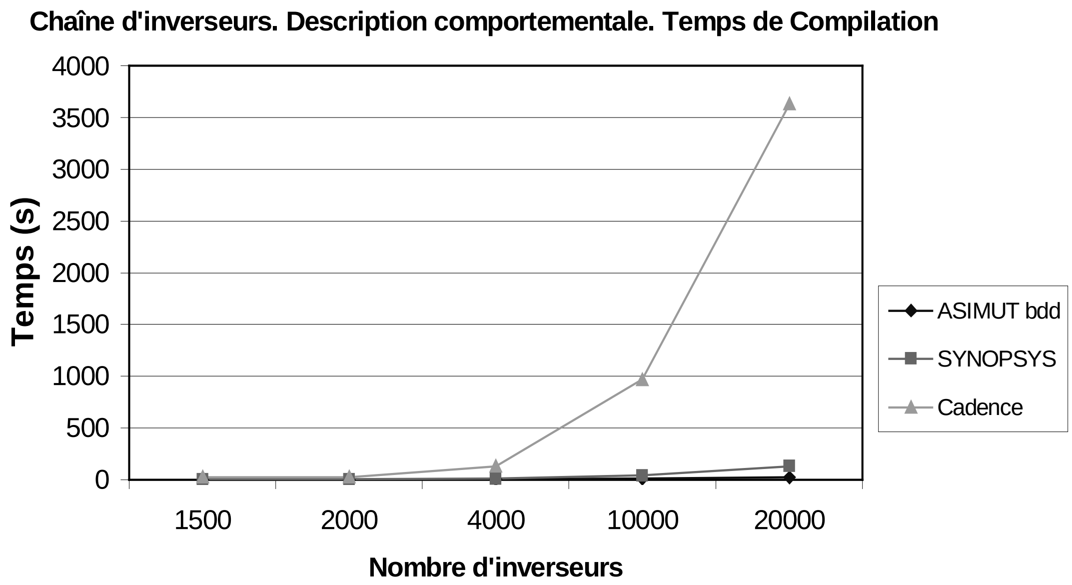
Asimut compile et exécute rapidement, ce qui lui permet de devancer Cadence et Synopsys.

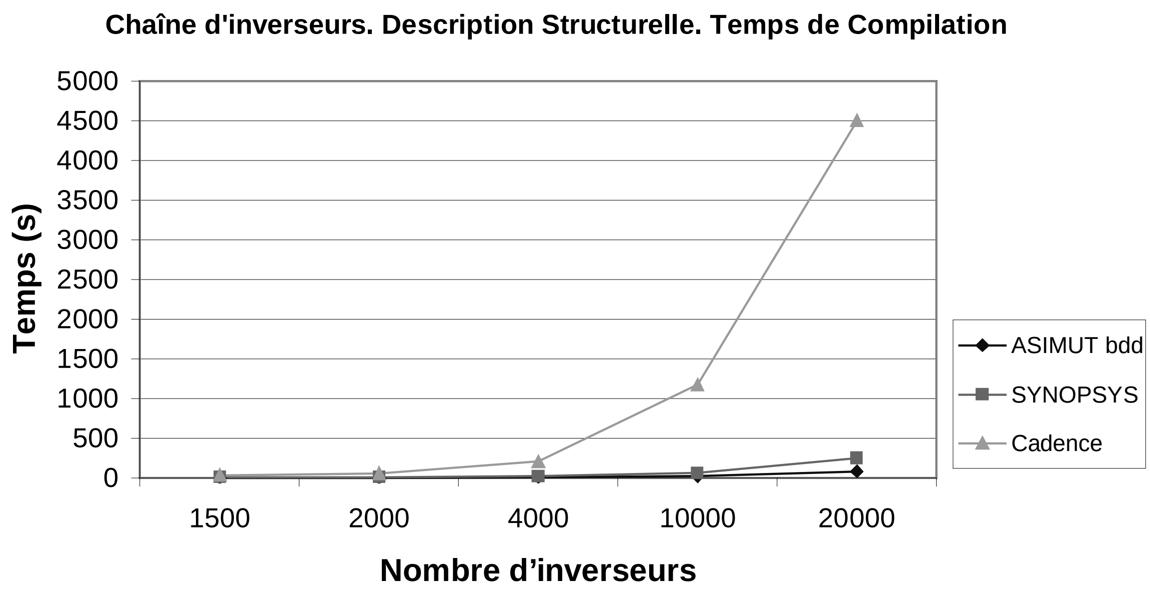




Cadence et Asimut ont des temps d'exécution du même ordre de grandeur. L'optimisation réalisée par Cadence lors de sa phase de compilation lui permet d'être 4 fois plus rapide que Synopsys en temps d'exécution.

Cadence est d'une version très récente (Février 1997), la version de Synopsys date elle de 1995. Deux ans d'écart sépare ces deux programmes, ce qui pourrait expliquer la différence de performance.





Asimut est le plus rapide lors de la compilation de ces benchmarks, Cadence et Synopsys qui sont des simulateurs compilés, ont pourtant des temps de compilation très différent. Ainsi Cadence compile jusqu'à 27 fois plus lentement que Synopsys qui lui même est jusqu'à 6 fois plus lent qu'Asimut.

Synopsys

Synopsys impose de compiler successivement et individuellement chacun des fichiers de la description. Nous garderons donc la mémoire maximum qui a été nécessaire lors de chaque compilation.

La mémoire nécessaire à l'exécution est directement disponible. Nous considérerons comme mémoire nécessaire à la simulation, la plus grande des deux quantités : mémoire de compilation et mémoire d'exécution.

Cadence

Cadence impose une phase d'élaboration avant de pouvoir exécuter un circuit.

La mémoire nécessaire que nous garderons pour la simulation par Cadence sera la plus grande des trois quantités de mémoire nécessaires : compilation, élaboration et exécution.

1.1) Benchmarks 1 et 2.

Dans un premier temps, nous nous sommes inspirés d'une étude de M. HUEBER [HUE91] publié en 1991, comparant les performances de 4 simulateurs VHDL.

Cette étude a consisté à simuler des inverseurs décrits de plusieurs manières :

BENCH 1 : Une première étude simule une description contenant n équations "flot de données" écrits : S_{i+1} = NOT S_i, il s'agit de la description comportementale d'une chaîne de n inverseurs.
BENCH 2 : Les mêmes circuits sont décrits de façon structurelle, sous forme de n instances d'une cellule NOT décrite en flot de données, (cf. BENCH1).

Nous refaisons cette étude en utilisant Asimut, Cadence et Synopsys. Nous avons simulés ces quatre benchmarks avec 5000 cycles d'horloge en entrée. Cette étude permet de mettre en évidence les qualités et capacités intrinsèques des simulateurs.

1.1.1) Temps de simulation

Cette étude a été menée avec les deux versions d'Asimut (Abl et Bdd), les ordres de grandeur étant identiques, nous avons préféré ne pas faire figurer l'étude d'Asimut-Abl afin de ne pas gêner la lisibilité des graphiques ci-dessous. On pourra se référer à l'annexe 4 de ce manuscrit afin de consulter les tableaux complets de l'étude.

Ces simulateurs sont dits "*full-VHDL*", ils acceptent l'ensemble des fonctionnalités du langage. Nous avons volontairement choisis de nous comparer à de tels simulateurs, car nous cherchons à montrer qu'en restreignant VHDL à un sous-ensemble (qui reste utilisable pour la conception de gros circuits), le simulateur est plus performant qu'un simulateur "*full-VHDL"*. Il nous faut donc comparer Asimut aux simulateurs *full-VHDL* afin de faire ressortir ce point.

Les mesures ont été réalisées sur la même station de travail : une station Sun Sparc 10, Processeur SuperSparc 50 (70 SPECint 92 ~ 70 Mips), 416 Mo de RAM, 400 Mo de swap sous SunOS 4.1.3.

Le programme GNU time a été utilisé sur plusieurs simulations consécutives, les graphes ci-dessous ont été réalisés avec la moyenne des résultats.

Tous les logiciels ont été utilisés en mode "*batch*", sans utiliser d'interface graphique et sans sauvegarde des résultats afin de ne mesurer que les performances intrinsèques des compilateurs et simulateurs VHDL.

Calcul des divers temps :

Asimut :

Asimut permet d'exécuter séparément la compilation. Le temps de compilation est donc directement fourni par le programme time. Nous avons donc soustrait le temps de compilation du temps total afin d'obtenir le temps d'exécution.

Synopsys :

Chacun des fichiers de la hiérarchie de la description doit être compilé successivement par un lancement du programme vhdlan. Il n'y a cependant pas de compilation séparée : toute modification dans un module oblige à recompiler toute la description. Nous avons donc additionné les temps de compilation de tous les fichiers afin d'obtenir le temps de compilation total. Synopsys permet d'exécuter directement une simulation à partir des fichiers issus de la compilation, nous obtenons donc directement le temps d'exécution. L'addition du temps de compilation et du temps d'exécution a pour résultat le temps de simulation.

Cadence

Cadence utilise la même méthode de travail que Synopsys. Nous additionnons donc les temps de compilation de chacun des fichiers afin d'obtenir le temps de compilation total.
Toutefois pour Cadence, il y a une étape d'élaboration, nous avons donc additionné le temps d'élaboration (programme ev) et le temps de simulation (programme sv) afin d'obtenir le temps d'exécution de Cadence.

Calcul de la mémoire nécessaire

Asimut

Asimut compile toute la hiérarchie de la description à simuler et indique la quantité de mémoire nécessaire afin de compiler l'ensemble de la description.

De plus, Asimut compile et simule en une seule exécution. Nous disposons donc pour Asimut de la mémoire totale nécessaire à l'exécution, c'est cette quantité qui sera utilisé dans les graphiques ci-dessous.

L'évaluation des performances d'un simulateur doit être réalisée selon plusieurs critères. Si le premier critère des utilisateurs reste le temps total de simulation, d'autres critères sont à prendre en compte. Par exemple, l'espace mémoire nécessaire à la simulation d'une description permet d'apprécier la quantité de ressources nécessaire à l'utilisation d'un simulateur.

De même, le temps de simulation n'est pas un critère suffisant, il faut différencier ses deux constituants : le temps de compilation et le temps d'exécution. Le temps de compilation est un critère important dans la phase de mise au point du modèle. Il est un des principaux facteurs du cycle simulation – analyse des résultats – mise au point.

La vitesse d'exécution est déterminante dans les phases de vérification finale où on simule des millions de cycle pour valider la conformité d'un modèle.

Dans ce chapitre, nous allons présenter une comparaison entre notre prototype Asimut et deux simulateurs du commerce : Cadence (simulateur Leapfrog) et Synopsys (simulateur VHDLsim). Cette comparaison est effectuée selon les quatre critères suivants : le temps de simulation, le temps de compilation, le temps d'exécution, et l'espace mémoire nécessaire à la simulation.

Enfin, nous comparons deux versions d'Asimut utilisant la première, une technique de représentation des expressions Booléennes à l'aide des BDDs et la seconde à l'aide des ABLs.

Les descriptions utilisés dans l'ensemble de cette comparaison sont : soit des cas d'école permettant de mettre en évidence le fonctionnement des simulateurs, soit des circuits réels.

1) Asimut et les simulateurs du commerce

Nous voulons comparer Asimut a deux simulateurs VHDL du commerce :

- Vhdlsim, le simulateur VHDL de Synopsys. (version 3.3a, 1995)
- Leapfrog, le simulateur VHDL de Cadence. (version 2.50-p001, Février 1997)

En effet, ces deux simulateurs selon une étude de 1995 [ROH95], sont parmi les plus rapides du marché, on trouve ci-dessous une comparaison de leurs performances :

	Synopsys V3.2	Synopsys V3.0a	IKOS V1.41	Model Tech V4.2	Vantage V5.0	Cadence Leapfrog V2.1
VSK	2.1	1.3	1.0	2.0	1.1	1
8051		4.4	3.9	2.5	4.6	1
LCD	1.1	3.5	2.7	1.7	3.1	1
BADGE		8.0	16.9	4.5	3.6	1
DIVI	6.3	17.9	8.1	5.7	7	1
INV1000	2.7	4.3	1.7	1.3	1.6	1
KOM	4.6	12.3	6.0	4.8	4.6	1
SIG	3.6	31.7		2.6	5.2	1
SPARC	1.0	11.8	1.8	2.2	3.3	1.3
Ø	3	10	5	3	4	1

Temps de simulation (relatif au simulateur le plus rapide). Source : [ROH95]

Chapitre VIII

Résultats et Performances

Nous avons présenté dans ce chapitre l'architecture de notre outil Asimut, des principes de conception et de réalisation jusqu'à la structure de données utilisée.

L'utilisation de cette structure de données a ensuite été expliqué et un exemple simple a permis de comprendre le déroulement d'une simulation.

La mise en œuvre a été réalisé selon les spécifications de départ : utilisation du VHDL, utilisation de structure de données statiques et représentation des expressions Booléennes à l'aide de graphes.

Mettre dans **EXECUTE** le signal **sout** qui dépend de **d**.

Pour **out** :

Valeur actuelle : 1, valeur projetée : 0

→ Valeur actuelle devient : 0, pas de dépendance sur le signal.

2nd delta-cycle : La pile **EXECUTE** contient le signal **sout**.

Pour **out** :

L'équation Booléenne de **sout** pointé par sa **wrxxx** est "i1 xor i2".

Remplacé par les instances de signaux pointé par la **lkdins** elle devient "**d** xor **c**".

La valeur projetée de **sout** est donc : 1 xor 0 = 1.

Mettre le signal **out** dans la pile **UPDATE**

Pour la pile **UPDATE** :

Pour **out** :

Valeur actuelle : 0, valeur projetée : 1

→ Valeur actuelle devient : 1, pas de dépendance sur le signal.

Fin du traitement du 2nd stimuli.

4.3) Cas des signaux gardés

En VHDL, les signaux résolus peuvent avoir plusieurs émetteurs. Lorsqu'un tel signal est gardé, chaque émetteur possède une condition de connexion/déconnexion aussi appelé *condition de garde*. Dans notre sous-ensemble, tous les signaux résolus sont gardés, ainsi à chaque signal résolu-gardé de la description, correspond une liste de bi-valeurs (valeur projetée de l'émetteur, valeur projetée de la condition de garde).

A chaque évaluation de ce signal, une *fonction de résolution* spécifique fournit la valeur du signal en fonction de la liste de ses bi-valeurs.

Dans notre sous-ensemble, nous avons défini trois fonctions de résolutions : une pour les signaux gardés de type *REGISTER*, et deux pour les signaux gardés de type *BUS OU-câblé ou préchargé*, et de type *BUS Multiplexé*.

Ainsi les signaux gardés sont traités comme les signaux non gardés, excepté leurs fonctions d'évaluation qui leur sont spécifiques, et qui sont prédéfinis dans le simulateur.

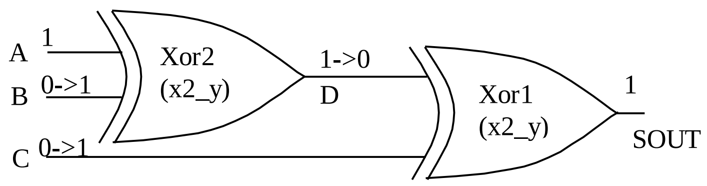
Ces fonctions interviennent lors de l'évaluation d'un signal gardé dans notre simulateur. Afin d'obtenir la valeur courante (ou actuelle) de ce signal, la fonction de résolution adéquate est appelée. Cette fonction de résolution reçoit comme paramètres la liste des bi-valeurs de chacun des émetteurs de ce signal, elle calcule et retourne alors la valeur courante du signal.

5) Conclusion

2ème delta-cycle : La pile **EXECUTE** est rempli avec **sout**.
Pour **sout** :
L'équation Booléenne de **sout** pointé par sa **wrxxx** est "i1 xor i2".
Remplacé par les instances de signaux pointé par la **lkdins** elle devient "**d** xor **c**".
La valeur projetée de **sout** est donc : 1 xor 0 = 1.
Mettre le signal **sout** dans la pile **UPDATE**

Pour **sout** :
Valeur actuelle : 0, valeur projetée : 1
→ Valeur actuelle devient : 1

Fin du traitement du 1er stimuli.



Deuxième Stimulus

Début du traitement du second stimuli.
Le **paseq** contient l'événement **b**=1 et **c**=1. (a n'a pas changé de valeur, il n'y a donc pas d'événement attaché à a).
Le signal **a** contient dans sa liste de dépendances le signal **d**.
Le signal **b** contient dans sa liste de dépendances le signal **d**.
Le signal **c** contient dans sa liste de dépendances le signal **sout**.

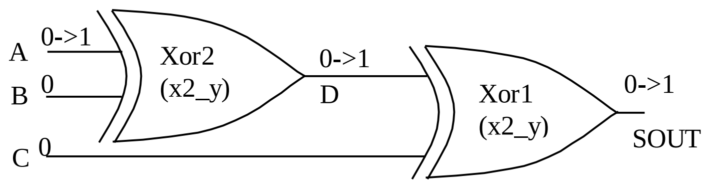
1er delta cycle : La pile **EXECUTE** est rempli avec **d** et **sout**.

Pour **d** :
L'équation Booléenne de **d** pointé par sa **wrxxx** est "i1 xor i2".
Remplacé par les instances de signaux pointé par la **lkdins** elle devient "**a** xor **b**".
La valeur projetée de **d** est donc : 1 xor 1 = 0.
Mettre le signal **d** dans la pile **UPDATE**

Pour **out** :
L'équation Booléenne de **sout** pointé par sa **wrxxx** est "i1 xor i2".
Remplacé par les instances de signaux pointé par la **lkdins** elle devient "**d** xor **c**".
La valeur projetée de **sout** est donc : 1 xor 1 = 0.
Mettre le signal **sout** dans la pile **UPDATE**

Pour la pile **UPDATE** :
Pour **d** :
Valeur actuelle : 1, valeur projetée : 0
→ Valeur actuelle devient : 0

Pour chaque pattern,
Pour chaque événement contenu dans la **paseq**
Remplir la pile **EXECUTE** avec les signaux dépendants de chaque événement.
Tant que la pile **EXECUTE** est non vide :
Pour chacun des signaux de la pile **EXECUTE**.
Evaluation de l’expression logique attaché à ce signal.
Stockage de l’évaluation dans le champ valeur projetée.
Mettre dans la pile **UPDATE**, ce signal.
Fin Pour
Pour chacun des signaux de la pile **UPDATE**
Comparaison entre valeur projetée et valeur actuelle.
Si les valeurs sont différentes, alors
Mettre à jour valeur actuelle
Mettre dans **EXECUTE**, les signaux dépendants du signal étudié.
Fin Si
Fin Pour
Fin Tant que
Sauvegarder l’état du circuit dans la **paseq**.
Fin Pour
Fin Pour



Premier Stimulus

Début du traitement du premier stimuli.
Le premier **paseq** contient l'événement **a**=1, (b et c n'ont pas changé de valeur).
Le signal **a** contient dans sa liste de dépendances le signal **d**.
Le signal **b** contient dans sa liste de dépendances le signal **d**.
Le signal **c** contient dans sa liste de dépendances le signal **sout**.

1er delta-cycle : La pile **EXECUTE** est rempli avec **d**.

Pour **d** :
L'équation Booléenne de **d** pointé par sa **wrxxx** est "i1 xor i2".
Remplacé par les instances de signaux pointé par la **lkdins** elle devient "**a** xor **b**".
La valeur projetée de **d** est donc : 1 xor 0 = 1.
Mettre le signal **d** dans la pile **UPDATE**

Pour la pile **UPDATE** :
Pour **d** :
Valeur actuelle : 0, valeur projetée : 1
Valeur actuelle devient : 1
Mettre le signal **sout** dans la pile **UPDATE**

stocker le nom du signal, sa valeur actuelle et projetée ainsi que le connecteur du modèle comportemental auquel il est associé.

Le circuit chip, (la racine de la description), va donner son nom à la structure de simulation, celle ci remplit les structures **simsig** et **bussig** avec les signaux de la description du circuit "circuit".

La structure contient deux instances Xor1 et Xor2. Celles-ci pointent toutes les deux sur la même structure de donnée comportementale représentant le comportement du modèle x2_y.

Ainsi pour l'instance Xor2 :

Le signal a connecté au signal d'entrée simple i1 du modèle x2_y,
Le signal b connecté au signal d'entrée simple i2 du modèle x2_y,
Sont représentés dans la structure **wrssig**. Nous obtenons ainsi 2 structures **wrssig**.

Le signal d connecté au signal de sortie simple t du modèle x2_y est représenté dans la structure **wrssig**, comme il s'agit d'un signal de sortie, la structure **wrssig** pointe sur le signal connecté au port, c'est à dire à la **simsig** contenant d.

Pour l'instance Xor1 :

Les signaux c, d sont représentés dans une structure **wrssig**.
La **wrssig** contenant la sortie sout pointe sur la **simsig** contenant sout.

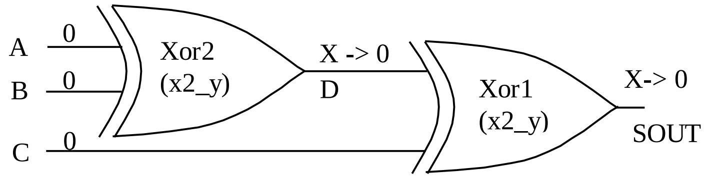
Enfin, la structure de donnée stimuli reçoit les pointeurs vers les signaux concernés de la structure de donnée simulation, évitant une indirection des pointeurs, afin d'accélérer la gestion de l'échéancier.

Etape V : Simulation de la description.

Chaque ligne du fichier de stimuli est représenté par un **paseq**, les **paseq** sont une liste chaînée ordonnée.

Chaque **paseq** représente un vecteur de stimuli, notre simulateur est à délai nul, on suppose donc que les vecteurs se succèdent les uns après les autres après le même délai.

Tous les signaux de la description sont initialisés à 0. En effet, la norme VHDL précise que tous les types VHDL ont une valeur d'initialisation par défaut. Cette valeur pour les types BIT et ses sous-types est la valeur du premier élément de l'énumération à savoir "0".



Initialisation du circuit

```
pat_1 : 1 1 1 ?1 ;
end;
```

Nous avons défini notre propre format de représentation des stimuli, en effet, VHDL permet de représenter les stimuli, mais pour cela, il est nécessaire d'utiliser les clauses temporelles du langage. Or, notre sous-ensemble exclut ces clauses. Le format de stimuli (ou pattern) que nous avons défini permet de représenter l'avancement du temps.

Ces valeurs correspondent au tableau de vérité suivant :

Signal	Pattern1	Pattern2
A	1	1
B	0	1
C	0	1
out	1	1

Nous avons maintenant l'ensemble des fichiers nécessaires à Asimut afin de simuler notre circuit. Nous allons expliquer comment Asimut simule ce circuit :

Etape I : Le fichier à simuler est structurel : Exécution du compilateur structurel sur le fichier *circuit.vst*.

Obtention d'une structure de données structurelle. Asimut examine la structure de données afin de connaître le niveau de hiérarchie de la description. Si le niveau de hiérarchie est supérieur à un, Asimut met à plat la description, ce n'est pas le cas ici. La structure de donnée structurelle contient les informations contenues dans le fichier de description structurelle, à savoir : Noms des connecteurs internes et externes, noms des modèles utilisés avec leur port d'entrée/sortie, et enfin description de la connexion des instances. Chacun des signaux est représenté par un champ dans la structure de données structurelle, ainsi chaque vecteur de signaux est dévectorisé. Par exemple : ***a(0 TO 31)*** serait représenté par 32 signaux, chacun des signaux étant représenté par le nom du signal suivi d'un espace suivi de l'indice du signal, par exemple : « ***a 21*** ».

Etape II : Les deux éléments de la hiérarchie sont Xor2 et Xor1. Asimut va exécuter le compilateur comportemental sur chacun des fichiers de description, en identifiant les dépendances des signaux dans la description, les expressions logiques sont représentés par des BDDs ou des ABLs. On obtient ainsi une structure de données comportementale pour chaque modèle. La structure de donnée comportementale contenant les informations contenues dans le fichier de description comportemental, à savoir : Noms des connecteurs et des signaux et représentation des équations logiques en ABLs ou en BDDs. Dans la structure de donnée structurelle, comme dans la structure de donnée comportementale, les vecteurs sont dévectorisés : à chacun des signaux du vecteurs est associée une équation logique.

Etape III : Traitement des stimuli : Exécution du compilateur de stimuli et obtention de la structure de donnée stimuli. La structure de donnée stimuli contient les informations contenues dans le fichier de description des stimuli appliqués au circuit, à savoir : les noms des connecteurs ou des signaux et les valeurs imposées, prévues ou demandées.

Etape IV : Création de la structure de simulation, les structures de données comportementales, structurelles et stimuli vont nous servir à créer la structure de simulation. La structure de donnée structurelle va être parcourue, et chaque instance d'un modèle va créer une **lkdins** (instance d'un modèle), cette **lkdins** va pointer vers la structure de donnée comportementale concernée. Chacun des signaux de l'instance possède sa structure de données permettant de

Description comportementale de la cellule XOR à 2 entrées :

```
-- VHDL data flow description of XOR_2
-- Entity Declaration

ENTITY x2_y IS
  PORT (
  i0 : in BIT; -- i0
  i1 : in BIT; -- i1
  t : out BIT; -- t
);
END x2_y;

-- Architecture Declaration

ARCHITECTURE behaviour_data_flow OF x2_y IS
BEGIN
t <= (i0 xor i1);
END;
```

Et la description structurelle du circuit complet est la suivante :

```
ENTITY circuit IS

PORT(
  a                    : in BIT;
  b                    : in BIT;
  c                    : in BIT;
  sout                  : out BIT;
);

END chip;

ARCHITECTURE fonctional OF chip IS

SIGNAL d               : BIT;

COMPONENT x2_y
PORT(
  i0 : in BIT; -- i0
  i1 : in BIT; -- i1
  t : out BIT; -- t
);
END COMPONENT;

BEGIN

Xor2 : x2_y
PORT MAP(
  i0 => a,
  i1 => b,
  t  => d,
);

Xor1 : x2_y
PORT MAP(
  i0 => d,
  i1 => c,
  t =>  sout,
);

end;
```

Pour simuler le circuit, il faut lui appliquer des stimuli, voici un exemple de fichier de stimuli :

```
in a;
in b;
in c;;
out;
begin

# a=1, b=0, c=0, we expect value 1 on out
pat_1 : 1 0 0 ?1 ;
# a=1, b=1, c=1, we expect value 1 on out
```

représente un délai nul pour la simulation, il représente la causalité engendrée par les diverses affectations. Un delta-cycle consiste à évaluer tous les processus qui sont dans la pile **EXECUTE**, chaque processus simulé agit sur un ou plusieurs signaux, leurs valeurs calculées et leurs index sont enregistrés dans la pile **UPDATE**. Quand la pile **EXECUTE** est vide, la pile **UPDATE** est dépilée en vérifiant si les signaux ont subis une modification. Si un signal a subi une modification, la valeur actuelle du signal est modifié et les objets qui dépendent de ce signal sont enregistrés dans la pile **EXECUTE**. Si un signal n'a pas subi de modification, aucune action n'est effectuée.

L'échéancier d'Asimut consiste donc en la structure **PASEQ**, ainsi que deux piles **EXECUTE** et **UPDATE**.

4) Déroulement d'une simulation dans Asimut

4.1) Algorithme d'Asimut

Données fournies en entrée : Nom du fichier de description, nom du fichier de stimuli.

Description structurelle

- Exécuter le compilateur structurel sur le fichier de description spécifié.
- Mettre à plat la description structurelle (*).
- Sur chaque fichier comportemental :
    - Exécuter le compilateur comportemental sur le fichier de description spécifié.
    - Identifier les dépendances dans la description et transformer les expressions Booléennes en BDD ou en ABL.
- Exécuter le compilateur de stimuli sur le fichier de description spécifié.
- Créer la structure de simulation LKDFIG en liant toutes les structures obtenues.
- Simuler la description :
  Tant que stimuli non vide
    - Lecture d'un stimuli.
    - Évaluation des instances connaissant un événement
    - Mise à jour de la valeur des signaux
    - Sauvegarde de l'état de la description
  Fin tant que

Dans le cas d'une description comportementale, nous créons une description structurelle qui contient une instance du modèle comportemental.

4.2) Simulation d'un exemple simple

Nous allons étudier un circuit structurel composé de deux cellules xor décrites de façon comportementale :

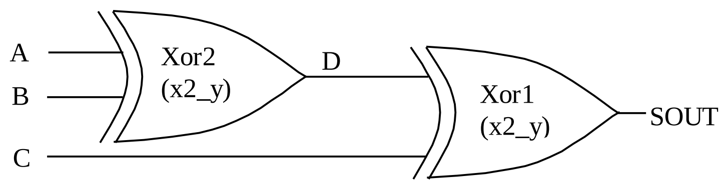


Fig. 7.4 : Exemple de circuit à simuler

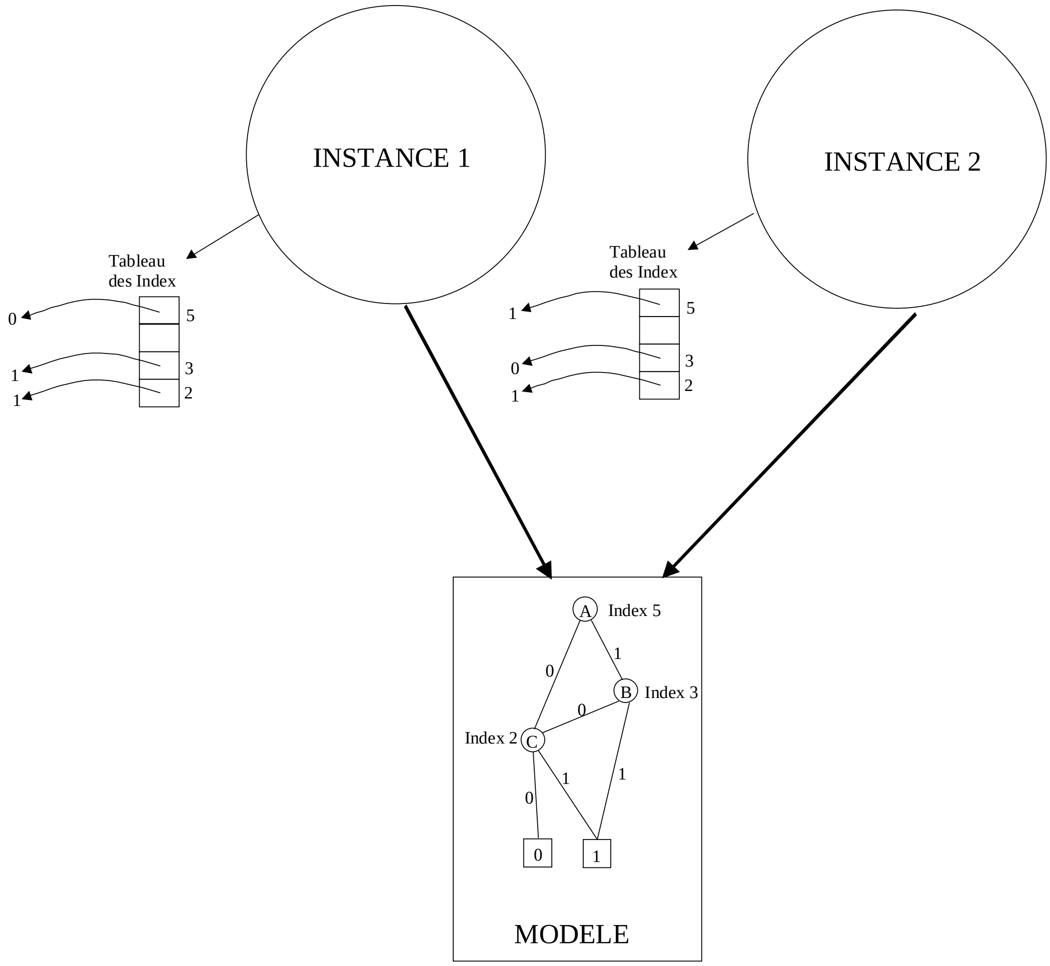


Fig. 7.3, Principe de la factorisation des modèles

3.3) L’échéancier

L’algorithme de simulation *event-driven* que nous avons choisi, utilise la propagation des événements à travers le circuit afin d’évaluer les signaux internes et de sortie. Les événements aux entrées du circuit sont définis par le fichier de stimuli.

Le système permettant de gérer la survenance des événements s’appelle un échéancier. Il permet d’enregistrer tous les événements à venir et de les restituer dans le temps.

Notre simulateur fonctionnant à délai nul (du point de vue de l'utilisateur, toutes les portes ont un temps de propagation nul), l’échéancier est donc simple. Il consiste en trois éléments, la liste ordonnée des stimuli (ou patterns) **PASEQ**, qui contient les stimuli appliqués aux entrées du circuit et deux piles de type FIFO : une pile des signaux à évaluer au temps actuel (**EXECUTE**) et une pile des signaux à mettre à jour (**UPDATE**) à la fin du delta-cycle.

Le delta-cycle est un concept défini par la sémantique de simulation du VHDL, il s'agit d'un cycle de simulation, consistant en une propagation des événements. Un delta-cycle

Ainsi le parcours d'un nœud d'un ROBDD ne nécessite que deux indirections : l'indice du nœud du ROBDD indique le déplacement dans le tableau et le contenu du tableau est le pointeur sur la valeur du signal. Suivant cette valeur, on prendra le fils gauche ou droit du nœud du ROBDD.

Une autre étape importante dans un simulateur logique et la propagation des événements dans le réseau. Chaque fois qu'un événement survient sur un signal, l'ensemble des nœuds du réseau dont au moins une entrée est reliée à ce signal doivent être réévalués. Afin d'économiser le temps nécessaire à l'identification de ces nœuds au cours de la simulation, nous construisons ces listes de dépendances de façon statique. Cette construction s'effectue dés l'obtention de la structure de simulation. Ainsi, à chaque nœud du réseau Booléen est associé sa propre liste de dépendance.

Chaque fois que la valeur du signal de sortie d'un nœud change, en parcourant la liste des dépendances, on accède directement aux nœuds qu'il faut réévaluer. Finalement, il reste à associer à chaque nœud du réseau, une valeur ('0' ou '1'). Elle représente la valeur du signal de sortie du nœud à chaque instant. L'ensemble de ces valeurs constitue l'état du circuit à un instant. Ainsi, dans notre simulateur, effectuer un pas de simulation consiste simplement à modifier les valeurs dans les structures qui représentent les nœuds du réseau Booléen.

3.2) Optimisation

Du point de vue implémentation logicielle, il est possible d'optimiser ce modèle théorique pour économiser de la place en mémoire en exploitant la multi-instanciation. Dans une net-list hiérarchique, les éléments terminaux de la hiérarchie sont souvent des instances multiples d'un même modèle. Par exemple, une "net-list" de portes logiques peut contenir plusieurs milliers de cellules, alors qu'il n'y a que quelques dizaines de modèles différents.

On décide donc d'attacher les graphes représentant les expressions Booléennes aux modèles et non aux instances. Cela signifie que seuls les étages les plus élevés de la hiérarchie structurelle sont mis à plat. En revanche, le découpage structurel de dernier niveau est conservé.

Soulignons qu'il s'agit d'une simple optimisation d'implémentation : L'algorithme de simulation event-driven continue à accéder directement aux signaux "internes" à une instance. Cette optimisation complique légèrement la structure de données du simulateur puisqu'il faut intégrer des informations décrivant la net-list et des informations décrivant les réseaux Booléens associés aux feuilles de la net-list.

La structure de données structurelle est détaillée dans [PET94].
La structure de données comportementale est détaillée dans l'annexe II.
La structure de données de simulation décrite ci-dessus est détaillée dans l'Annexe III.

Les modules composant Asimut sont :
- Un compilateur VHDL de descriptions structurelles hiérarchiques.
- Un compilateur VHDL de descriptions comportementales.
- Un compilateur de fichier de stimuli.
- Un éditeur de liens qui va construire la structure de données de la simulation.
- Un noyau de simulation utilisant l'algorithme *event-driven*.
- Un "driver" de fichier de stimuli résultats.

3.1) Organisation des structures de données de simulation

Asimut accepte en entrée une description structurelle hiérarchique dont les "feuilles" sont des descriptions comportementales de type flot de données. Il a également besoin de la description des stimuli pour pouvoir effectuer une simulation. Or les structures de données génériques issues de la compilation de ces descriptions ne sont pas forcément adaptées à l'algorithme de simulation que nous voulons mettre en œuvre. Nous avons donc défini un ensemble de structures de données propre à Asimut, dites structures de simulation.

A l'issue de la phase de compilation, les structures génériques sont traduites en structures de simulation. Ces dernières représentent fondamentalement un réseau Booléen à plat : Les réseaux Booléens représentant les "feuilles" de la structure hiérarchique ont été fusionnés pour constituer un unique réseau Booléen représentant le circuit complet. Chaque nœud de ce réseau Booléen représente une assignation concurrente et est considéré comme un processus possédant un seul signal de sortie s'exécutant sous le contrôle de l'algorithme "event-driven".

A chaque nœud du réseau Booléen est attaché une expression Booléenne représentée par un ROBDD ou par un ABL, suivant le mode de simulation. Seuls les éléments mémorisants et les bus échappent à cette règle. A un élément mémorisant ou à un bus sont attachées deux expressions logiques : une représentant la valeur à écrire dans l'élément mémorisant, l'autre la condition d'écriture.

La performance d'un simulateur logique dépend énormément de la vitesse d'évaluation des expressions. Afin d'accélérer cette évaluation, chaque nœud de notre réseau Booléen peut accéder directement aux valeurs des signaux dont il dépend à travers un tableau de pointeurs qui lui est associé.

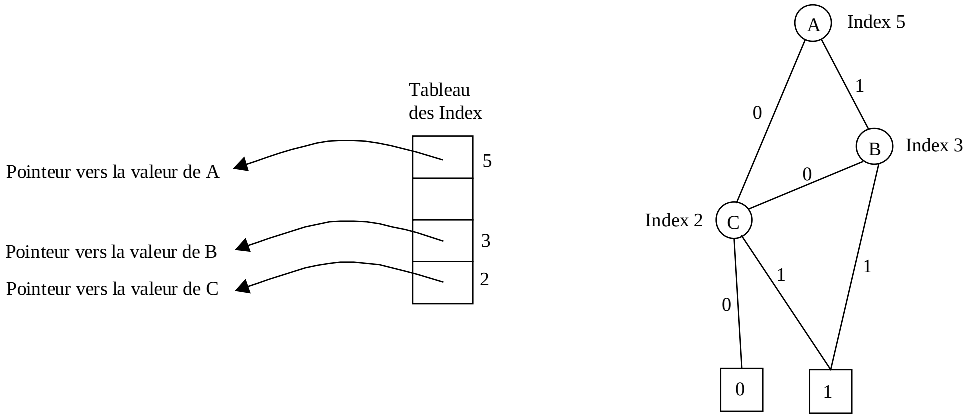


Fig. 7.2, Principe d'utilisation des ROBDD

Asimut fait partie de la chaîne de CAO Alliance développé par l'équipe Architecture du laboratoire LIP6. Cette chaîne couvre la totalité des étapes de conception d'un circuit intégrés numériques.

La diversité des formats de représentation de l'information est un des aspects les moins satisfaisants pour le développeur d'application, ces formats sont souvent liés à un environnement de conception spécifique. Alliance propose une solution à ce besoin d'uniformisation par le biais d'une plate-forme de développement pour outils de CAO. Cette plate-forme repose sur une structure de données orientée objet destinée à fournir un modèle non ambigu des différentes vues d'un circuit : vues physiques, structurelles et comportementales [PET94]. Des bibliothèques de fonctions relatives aux structures de données représentant en mémoire les différentes vues d'un circuit, permettent aux concepteurs d'outils de s'affranchir des problèmes liés à la lecture et à l'écriture des différents formats de fichiers utilisés industriellement (EDIF, SPICE, par exemple).

3) Architecture du simulateur

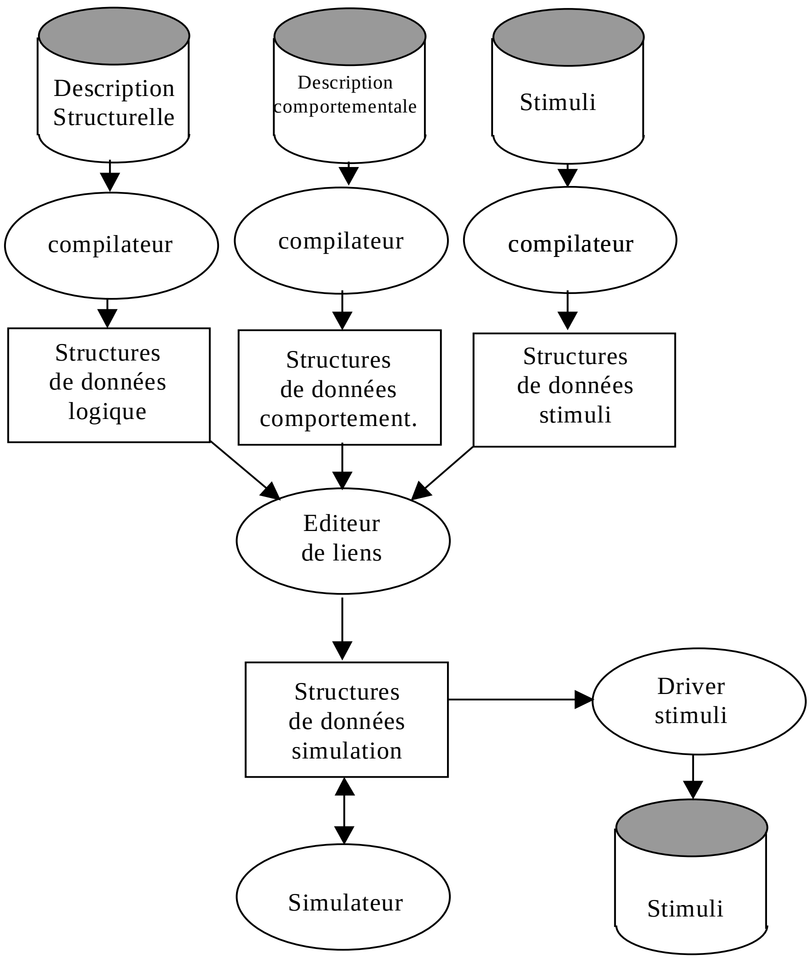


Fig. 7.1 : Architecture générale du simulateur

La nouvelle méthode de simulation que nous avons proposée au chapitre 5 a été mise en œuvre au sein d'un prototype appelé Asimut.

Le véritable enjeu de notre simulateur est l'utilisation de structures de données statiques. Nous allons montrer dans ce chapitre que cette technique nous permet de simuler un circuit en ne changeant que les valeurs des signaux à l'intérieur des structures de données. En effet, pour parvenir à réaliser un simulateur efficace, les structures de données sont construites une fois pour toutes, l'organisation de ces structures doit être telle que le nombre d'indirections nécessaires afin de retrouver une information soit minimal.

L'organisation de ces structures est également influencée par le choix de l'algorithme de simulation event-driven. Il faut pouvoir rapidement évaluer une expression et propager les événements vers les signaux en aval.

Nous présentons dans ce chapitre la réalisation effective d'un prototype logiciel de notre méthode de simulation appelé Asimut.

Nous décrivons l'architecture de ce prototype en exposant les idées directrices de l'organisation des structures de données, ainsi que les étapes importantes de la construction de ces structures de données.

Enfin, le déroulement d'une simulation sera expliqué grâce à un exemple simple, permettant de suivre l'utilisation des structures de données ainsi que la mise en œuvre de l'algorithme de simulation event-driven.

1) Méthodologie de développement

Nous avons décidé d'utiliser une méthodologie de conception classique de type « *descendante* » (ou Top-Down), cette méthodologie consiste à fragmenter le problème jusqu'à arriver à des modules distincts facilement compréhensibles, à fonctionnalité et interfaces parfaitement définis. Lors de la conception d'Asimut, nous avons décrit le diagramme des flux d'information à l'intérieur du programme. Ce découpage de l'architecture d'Asimut permet de le considérer comme un assemblage de modules facilement modifiables.

Ce découpage en *modules* nous permet de pouvoir développer de manière parallèle ces éléments. De plus, cela nous permet de faciliter la mise au point du simulateur, puisqu'il limite la portée des erreurs. Pour effectuer ce découpage en modules, il est nécessaire de spécifier parfaitement l'interface de communication entre les modules. En effet, pour modifier ou changer un des nouveaux modules, il suffit de respecter cette interface de communication afin de s'assurer que ce module s'insérera parfaitement dans le simulateur.

Nous avons mis en œuvre un principe issu de la *programmation orienté objet* : cette approche considère que chacun des modules composant Asimut possède des données *publiques* et des données *privées*. Ainsi le programme principal ne peut utiliser que les données *publiques* de chaque module, les données *privées* restant inaccessibles. Par cette approche, il est possible de modifier un module en veillant à ce que les données *publiques* conservent la même sémantique.

Asimut a été entièrement programmé en Langage C, les compilateurs ont été écrits avec des outils Unix standards : YACC (analyseur syntaxique) et LEX (analyseur lexical), grâce à l'utilisation de ces outils standards, notre simulateur est portable sur la majorité des plates-formes UNIX.

Chapitre VII

Architecture du simulateur

transcription directe des expressions Booléennes contenues dans la description du circuit à simuler, leur taille dépend de façon linéaire de la taille du fichier VHDL source.

L'évaluation des BDDs est un simple parcours de graphe, alors que l'évaluation des ABLs nécessite de définir une fonction d'évaluation pour chaque type d'opérateurs.

Il est donc difficile à priori de privilégier l'une ou l'autre méthode. Il nous semble préférable de les expérimenter toutes les deux et de les confronter. C'est un des objectifs de nos travaux.

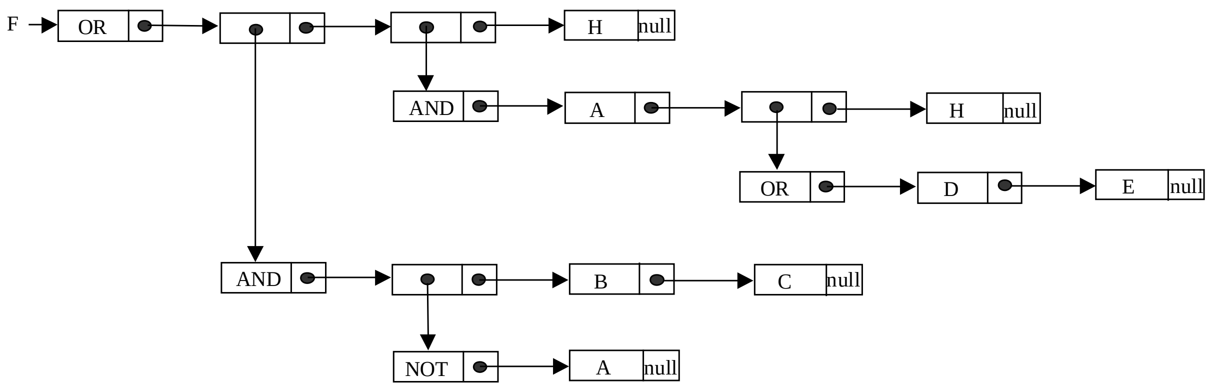


Fig. 6.2 : ABL de F= $\overline{A}BC + A.(D+E) + H$

Les ABLs ont la particularité d'être simple à construire. En effet, la structure d'un arbre ABL reflète fidèlement la structure de l'expression Booléenne qu'il représente.

L'évaluation des ABLs est plus longue que celle des BDDs, elle consiste en un parcours récursif de l'arbre, une fonction C étant associée à chaque type d'opérateur.

3) Domaines d'utilisation

Les BDDs sont utilisés depuis longtemps, dans des domaines comme la preuve formelle ou la synthèse logique. Le seul problème lors de l'utilisation des BDDs provient de la sensibilité des BDDs à l'ordonnancement des variables lors de la construction. En effet, cet ordonnancement a un impact direct très important sur l'encombrement en mémoire du graphe.

L'ordonnancement des BDDs est pour cette raison un domaine de recherche très actif dans le monde de la CAO [BRY94]. Néanmoins, cet ordonnancement nécessite des temps de calcul qui peuvent devenir très important.

Les arbres ABL sont facile à construire à partir du VHDL source, et il n'y a pas de risque d'explosion de l'encombrement mémoire. En revanche, l'évaluation directe d'un arbre ABL est moins directe que celle d'un graphe BDD. Nous avons choisi dans le cadre d'une simulation utilisant les BDDs de d'abord construire les ABLs, puis en utilisant les ABLs en mémoire de construire les BDDs.

4) Conclusion

Nous avons présenté dans ce chapitre deux méthodes de représentation d'expressions Booléennes : les BDDs et les ABLs.

La taille d'un graphe BDD dépend étroitement de l'ordonnancement des variables. Avec un ordonnancement sous-optimal, l'occupation en mémoire peut être très importante voire impossible. Les ABLs occupent beaucoup moins de place, en effet, comme ils sont la

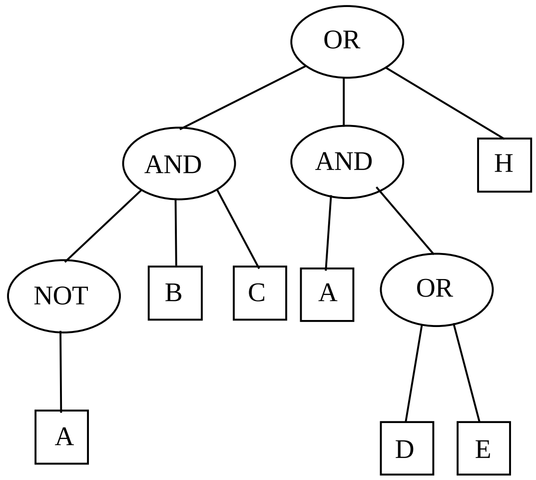


Fig. 6.1 : Représentation d'une équation en préfixé

Notre laboratoire d'accueil a développé une technique de représentation et de manipulation des expressions Booléennes parenthésées multi-niveaux. Cette technique s'appuie sur une bibliothèque de fonctions C nommée ABL (Arbres Binaires Lisp-like) [BUR93]. Un ABL est une représentation préfixée d'une expression Booléenne utilisant les opérateurs logiques : OR, NOR, NAND, XOR, AND et NOT.

Les ABLs sont mis en œuvre en utilisant des listes chaînées. Ces listes chaînées sont construites au moyen d'une unique structure de donnée qui est un bipointeur :

Structure de données des ABLs en langage C :

```
typedef struct chain   /* pointer list, lisp lists    */
   {
   struct chain *NEXT;    /* next pointer                */
   void         *DATA;    /* generic pointer             */
   }
chain_list;
```

Une expression atomique est faite d'un simple doublet ayant le pointeur NEXT égal à NULL et le champ DATA égal au pointeur sur une chaîne de caractères représentant le nom de la variable ou les constantes '0' et '1'.

Une expression opérateur est une liste d'objets, le premier objet est l'opérateur de tête de l'expression et les objets suivants sont les opérandes de l'expression. Une opérande peut être récursivement une expression atomique ou une expression opérateur.
Exemple d'utilisation des ABLs :

Soit l'équation F = $\overline{A}BC + A.(D+E) + H$, son ABL est représenté ci-dessous :

1.3) Application à la simulation

Si un BDD est créé pour chacun des nœuds du réseau Booléen, évaluer la valeur de l'expression Booléenne (et donc de la variable ou du signal qui lui est rattaché), consiste a parcourir le graphe en cheminant par les fils droits ou gauches selon la valeur du nœud jusqu'à arriver à un nœud terminal, '0' ou '1'.

Si le support d'une expression Booléenne contient n variables, l'évaluation de cette expression coûte au plus n indirections en mémoire..

1.4) Structure de données

structure d'un nœud de BDD en langage C :

```
typedef struct node
    {
    struct node *high,*low;
    short index ;
    short mark;
    } *pNode;
```

Les pointeurs **high** et **low** sont les fils droits et gauches.
index est l'index du signal, cet index est unique et identifie de manière canonique le signal.
mark donne le nombre de pères pointant sur le nœud.

Du point de vue de l'implantation, on associe un index à chaque variable d'entrée de la fonction Booléenne représentée par un graphe de décision binaire. Le nœud terminal '0' possède l'index 0, et le nœud terminal '1' possède l'index 1. Les index supérieurs à 1 sont réservés aux variables d'entrée de la fonction Booléenne représentée par le graphe, en respectant la règle qu'un nœud doit toujours pointer sur des nœuds fils ayant des index plus petits.

2) Les ABLs (Arbres Binaires Lisp-like)

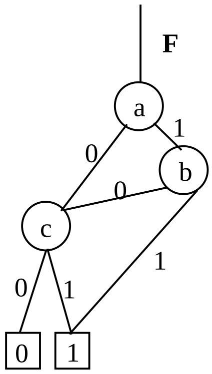
Une expression Booléenne parenthésée multi-niveaux peut toujours être écrite sous forme préfixée comme en Lisp [GIR85] [ALL78] :

Exemple : $\overline{A}BC + A.(D + E) + H$
s'écrit (OR (AND (NOT A), B, C), (AND A, (OR D, E)), H)

Cette expression peut toujours être représentée graphiquement par un arbre :

Un arbre de décision binaire peut être encore réduit pour produire un graphe de décision binaire. Cette réduction est obtenue en regroupant les sous-arbres isomorphes[BRY86].

L'arbre de décision binaire de l'exemple précédent peut conduire au graphe de décision de la figure ci-dessous.

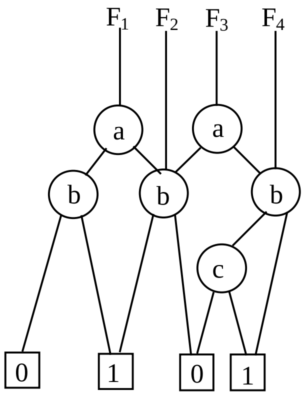


Un ensemble de fonctions Booléennes peut être représenté par un graphe analogue.

1.2) Multi-graphes de décision binaire

Un multi-graphe de décision binaire (ou multi-ROBDD) est un graphe orienté permettant de représenter un ensemble d'expressions Booléennes possédant le même support.

Exemple : Soit les fonctions Booléennes suivantes :



$$F_1 = a.\overline{b} + \overline{a}.b$$
$$F_2 = \overline{b}$$
$$F_3 = \overline{a}.\overline{b} + a.(b + c)$$
$$F_4 = b + c$$

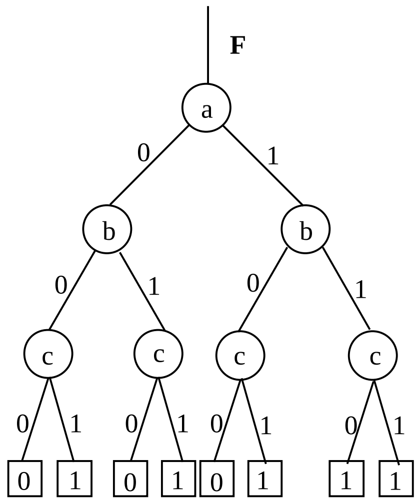
Le multi-graphe de décision binaire ci-contre peut être construit avec l'ordonnancement a, b et c.

Un réseau Booléen peut être représenté par un multi-ROBDD. Dans ce cas, l'ensemble des "variables" constituant le support commun est constitué par l'ensemble des signaux d'entrées plus l'ensemble des signaux intermédiaires du réseau Booléen, c'est à dire l'ensemble des variables qui apparaissent dans les membres de droite des expressions.

L'ensemble des racines du multi-ROBDD est constitué par l'ensemble des signaux de sortie, plus l'ensemble des signaux intermédiaires du réseau Booléen, c'est à dire l'ensemble des variables qui apparaissent dans les membres de gauche des expressions.

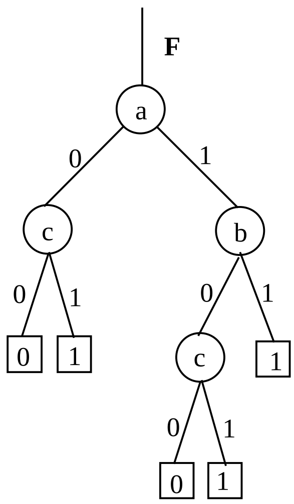
Chaque nœud non terminal **n** est associé à une variable d'entrée x_i. Il correspond à une certaine fonction F_n, et possède deux fils ou arêtes correspondant à $F_n(x_i=0)$ (arête gauche) et $F_n(x_i=1)$ arête droite.

Exemple : Soit la fonction $F = a.b + \overline{a}.c + \overline{b}.c$, la figure ci-dessous donne l'arbre de décision binaire de F selon l'ordonnancement a, b et c.



Un arbre de décision binaire est une représentation canonique (à ordonnancement fixé des variables du support) d'une fonction Booléenne. Pour prouver l'équivalence de deux fonctions Booléennes, il suffit de créer deux arbres de décision binaires et de parcourir conjointement leurs feuilles terminales en testant l'égalité des fonctions constantes.

La taille d'un arbre de décision binaire peut être réduite pour donner un arbre de décision binaire minimal. Cette simplification s'obtient en éliminant les nœuds redondants. Un nœud est redondant si toutes les feuilles terminales accessibles par ce nœud dénotent la même fonction constante. L'arbre de décision binaire minimal de l'exemple précèdent est donné par la figure ci-dessous.



A chaque nœud d'un réseau Booléen est attachée une expression Booléenne. La simulation d'une telle description revient finalement à évaluer ces expressions Booléennes au bon moment et avec les bonnes valeurs.

Nous avons expérimenté deux méthodes de représentation d'expressions Booléennes : les diagrammes de décision binaires également appelés *BDD*, et les arbres binaires ABL, permettant de représenter des expressions parenthésées multi-niveaux.

1) Les BDDs

Les Graphes de décision binaire ou (*Binary Decision Diagram*) seront appelés BDD. En effet, ce terme fait maintenant partie des termes informatiques courants, et nous avons préféré conserver cet anglicisme pour tout notre manuscrit.

Même si l'idée de base existe depuis plus de 30 ans [AKE78], c'est Randall BRYANT qui décrivit le premier une représentation canonique des BDDs [BRY86] et les premiers algorithmes avec mise en œuvre [BRA90].

La majorité des applications utilisant les BDDs appartiennent aux domaines de la preuve formelle et de la synthèse logique [BDD97] [BUR93], où la manipulation d'expressions logique est très importante et où la canonicité des BDDs est très utilisée.

1.1 Définition

Soit F une fonction Booléenne, F peut être décomposée par la formule de Shannon et selon la variable d'entrée x_i de façon suivante :

$$F(x_1, x_2,\dots, x_n) = \overline{x_1}.F(0, x_2, ..., x_n) + x_1.F(1, x_2, ..., x_n)$$

On note plus simplement :

$$F(0, x_2, ..., x_n) = F(x_1=0) \text{ et } F(1, x_2, ..., x_n) = F(x_1=1)$$

Cette décomposition peut être effectuée de manière récursive selon un ordre donné des variables d'entrée jusqu'aux fonctions constantes 0 et 1.

Exemple : Soit $F = a + b$, on peut écrire

$$F = \overline{a}.F(a=0) + a.F(a=1)$$
$$F = \overline{a}.(b.F(a=0,b=0) + b.F(a=0,b=1)) + a.(\overline{b}.F(a=1,b=0) + b.F(a=1,b=1))$$
$$F = \overline{a}.(\overline{b}.0 + b.1) + a.(\overline{b}.1 + b.1)$$

Arbres de décision binaire : Un arbre de décision binaire (ou arbre de Shannon) est une arborescence relative à la décomposition d'une fonction Booléenne par la formule de Shannon selon un ordre fixé de ses variables d'entrée [MOR82] [SAR95].

Propriétés de l'arbre de décision binaire :

Il existe un nœud spécifique appelé **nœud racine**.

Ses feuilles terminales représentent les fonctions constantes F = 0 et F = 1.

Chapitre VI

Représentation des expressions Booléennes

Ce qu'on peut espérer avec cette approche est d'obtenir un temps d'exécution nettement plus court que les temps d'exécution des simulateurs interprétés, en effet, l'évaluation d'une expression Booléenne, en utilisant notre technique, est ramené à un parcours de graphe, alors que dans un simulateur interprété une machine virtuelle doit interpréter un pseudo-code. De plus, nous voulons conserver des temps de compilation comparables ou même meilleurs, pour cela nous avons choisi de travailler sur un sous-ensemble du langage VHDL. De cette façon, nous diminuons la complexité du compilateur et par conséquent nous diminuons les temps de compilation nécessaires.

est stabilisé, sinon il faut continuer la propagation des événements, donc en évaluant de façon itérative les réseaux Booléens dépendants de ces nouveaux événements et ainsi de suite jusqu'à stabilisation du circuit.

Mais la taille des réseaux Booléens pouvant devenir trop importante pour être contenue dans la mémoire des ordinateurs. Nous décidons de factoriser chacun des réseaux Booléens représentant une description comportementale. De cette façon, nous limitons l'encombrement en mémoire de ces réseaux.

Le développement d'un simulateur de ce type est simple, en effet, la construction d'un réseau Booléen à partir d'expressions Booléennes est une technique connue. Le compilateur VHDL doit construire ces structures de données comportementales et structurelles. Le simulateur doit ensuite mettre à plat ces structures de données afin d'obtenir une unique structure de données consistant en un ensemble de réseaux Booléens. La simulation proprement dite consistera à parcourir les réseaux Booléens en évaluant les expressions.

L'utilisation d'un tel simulateur reste simple et didactique, chaque description est compilée puis simulée. En cas de changement à l'intérieur d'une description, celle ci est de nouveau compilée puis simulée. Nous nous attendons à ce que cette étape de compilation soit plus rapide que les étapes de compilation et d'édition de liens d'un simulateur compilé.

L'implémentation des fonctionnalités de mise au point de description dans un tel simulateur est aisée. L'utilisation de réseaux Booléens permet la conservation de chacun des signaux de la description à l'intérieur même de la structure de données de simulation en tant que variable d'un réseau. De plus chaque instance de modèle est identifiée par une mise en facteur unique d'un réseau Booléen, ainsi les points d'arrêts et les points d'observation sont facilement identifiables et exploitables.

La vitesse d'exécution reste la principale inconnue, c'est afin d'obtenir des résultats précis dans ce domaine que nous avons choisi d'expérimenter cette technique dans notre simulateur Asimut.

4) Conclusion

Nous avons présenté une technique utilisant des graphes pour représenter les nœuds du réseau Booléen constituant le circuit à simuler, de cette façon l'évaluation d'un des signaux de la description est ramené à un parcours de graphe.

Nous avons voulu tester cette technique de simulation afin de valider ce que nous pressentions, à savoir que la manipulation de réseaux Booléens pouvait être efficace afin de représenter et de simuler la description d'un circuit numérique.

Il est évident que l'utilisation de cette technique repose sur une utilisation adéquate des méthodes de représentation d'expressions Booléennes. Il est, en effet, nécessaire de parvenir à évaluer rapidement les expressions logiques décrivant le circuit à simuler. Mais il est également nécessaire de parvenir à construire rapidement l'ensemble des graphes représentant le comportement du circuit à simuler.

Nous autorisons dans notre sous-ensemble structurel les descriptions hiérarchiques de niveau multiples contrairement aux descriptions comportementales limités à un niveau de hiérarchie. Nous pouvons donc considérer une description VHDL comme un arbre ayant de multiples branches (description structurelle) dont les feuilles sont des réseaux Booléens (descriptions comportementales).

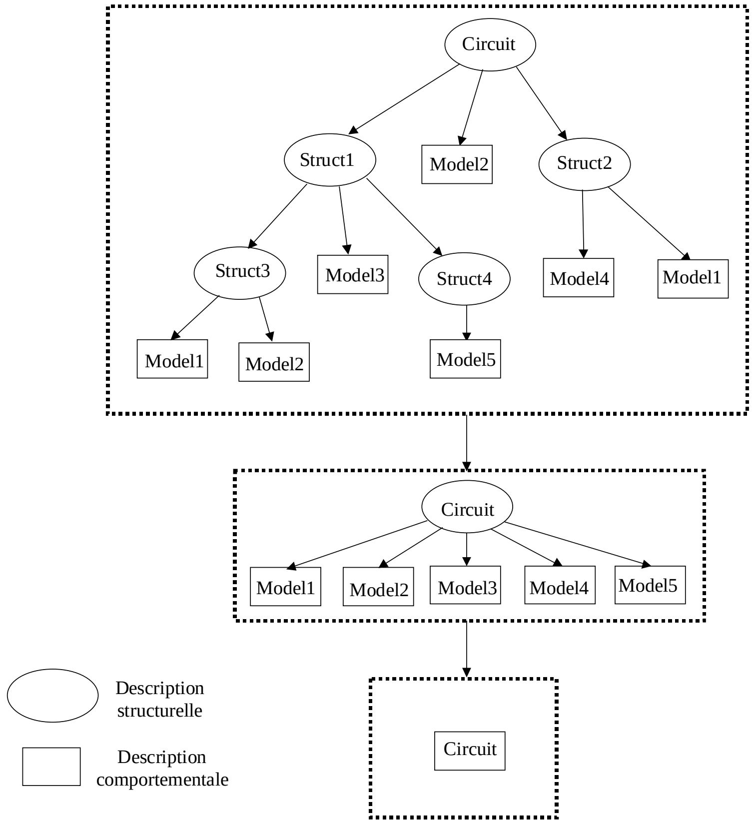


Fig. 5.1, Réduction des structures de données

La structure de donnée en mémoire représentant cette description aura la même hiérarchie. Les descriptions structurelles seront représentées dans une structure de données structurelles et les descriptions comportementales seront représentées dans une structure de données comportementales. Cette structure de données comportementale contiendra les réseaux Booléens représentant les expressions Booléenne de la description comportementale.

La structure de données structurelle ne contiendra que les noms des instances et des modèles et une liste d'association entre les connecteurs instanciés et les connecteurs virtuels du modèle.

Nous désirons réduire cet arbre à une unique structure de données comportementales lors de la simulation de la description.

Ainsi, nous nous retrouvons avec un ensemble de réseaux Booléens. La simulation de la description revient donc à évaluer le réseau Booléen en le parcourant. L'utilisation de l'algorithme *event-driven*, est bien adapté à cet ensemble de réseaux Booléens. Chaque événement n'intervenant que sur certains réseaux, il suffit de n'évaluer que les réseaux dépendants des événements en cours, si cette évaluation ne produit aucun événement le circuit

A l'exécution, les simulateurs compilés donnent des résultats 10 à 100 fois plus rapide que les simulateurs interprétées [MAU90][WAN90], il s'agit d'un résultat avéré et prouvé qui explique la préférence des industriels pour cette technique [SIN91].

Le temps nécessaire au développement d'un modèle VHDL est plus long avec un simulateur compilé. En effet, avec ces simulateurs, si une modification est faite dans le modèle, une nouvelle traduction doit être réalisé vers le langage C, ainsi que la compilation du code généré et l'édition de liens afin de reconstruire un simulateur spécifique. Ces opérations ne sont pas immédiates et augmentent le temps de mise au point, d'autant plus que la performance du simulateur dépend des phases d'optimisation du compilateur.

Dans un simulateur interprété, la seule étape pour arriver jusqu'à la simulation proprement dite est la génération d'un pseudo-code spécifique à la machine virtuelle, la simulation peut ensuite être lancée.

Développement

Le générateur de pseudo-code dans un simulateur interprété est généralement réalisé rapidement, puisqu'une grande latitude est laissée au développeur dans la conception de la machine virtuelle et du jeu d'instruction. La machine virtuelle doit néanmoins posséder des fonctions de contrôle parallèle, le développeur étant libre des techniques à mettre en œuvre.

Dans un simulateur compilé, le générateur de programme C doit se contenter des fonctionnalités offertes par le langage. Pour certaines fonctionnalités de VHDL et notamment le parallélisme, cette génération exige la création de bibliothèques de fonctions évoluées et spécifiques. Ces fonctions, sont par exemple, la sauvegarde et la restauration du contexte des processus. En effet, la sémantique du langage C est trop éloignée de la sémantique du VHDL pour permettre une traduction rapide. Tout ceci implique l'écriture complète d'un système multitâche en langage C.

C'est face à ces points que nous allons devoir comparer la solution que nous proposons.

3) Une troisième voie

Au lieu de représenter la description par un pseudo-code exécuté par une machine virtuelle. Nous avons choisi de représenter la description par une structure de données entièrement présente en mémoire.

Cette structure de données est obtenue par une mise à plat de la hiérarchie afin d'obtenir un réseau Booléen "a plat" représentant le circuit à simuler. Chaque nœud du réseau Booléen représente une assignation concurrente qui est considérée comme un processus s'exécutant en parallèle avec tous les autres.

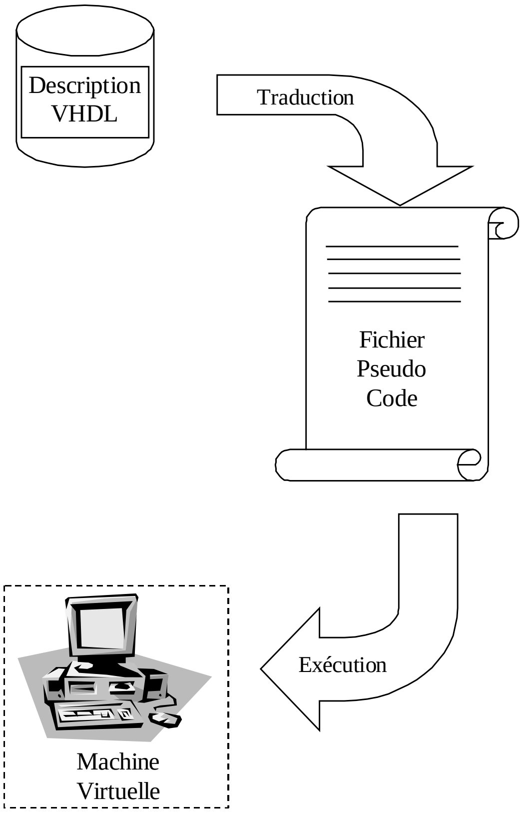
Chacun de ces processus est représenté par un graphe en mémoire décrivant l'expression Booléenne associée au membre de droite de l'assignation. L'évaluation d'un processus se réduit donc à un parcours de graphe.

La réduction du VHDL à deux sous-ensembles du langage, un sous-ensemble structurel et un sous-ensemble comportemental a été dicté par notre volonté de différenciation des deux types de description dans notre méthode de développement descendante (*top-down*).

L'exécutable finalement obtenu est donc spécifique à chaque description, ce qui signifie qu'à chaque modification d'un modèle de cette description, ce processus doit être répété afin d'obtenir un nouvel exécutable.

1.2) Techniques interprétées

Dans un simulateur interprété, chaque processus est traduit en un code intermédiaire, qui est ensuite exécuté sur une machine virtuelle.



Dans la suite du manuscrit, le "modèle interprété" fera référence à un système qui réalise toutes les analyses statiques, et produit un pseudo-code pour une machine virtuelle de bas niveau.

On appelle **machine virtuelle** ou **émulateur** un programme simulant un microprocesseur et son jeu d'instructions spécifiques.

Cette machine est conçue afin d'être la plus performante pour un environnement de simulation. Elle doit donc comporter des dispositifs de sauvegarde et de restauration de contexte, lui permettant de simuler très facilement une description parallèle, telle que celle d'un circuit logique.

Ses caractéristiques ayant été spécialement définis pour la simulation, son jeu d'instructions est entièrement dédié à cette tâche. Des ressources matérielles spécifiques sont également utilisées conjointement : mémoire, registres et fonction d'entrée-sortie.

Un échéancier central est essentiel pour pouvoir utiliser la sémantique complexe des instructions temporelles du VHDL. Les modules peuvent ainsi être évalués en temps voulu par un appel de l'échéancier.

<u>2) Comparaison entre réalisations compilées et réalisations interprétées :</u>

<u>Vitesse d'exécution</u>

A première vue, la simulation par la technique interprétée ressemble beaucoup à la technique compilée.

Elle nécessite également un échéancier pour contenir la variation des valeurs des signaux dans le temps. De même, chaque processus de la description est traduit en un code assembleur. La différence, et elle est de taille, réside dans le fait que le code ainsi obtenu n'est pas exécuté par une machine matérielle mais virtuelle.

Le VHDL permet de décrire le parallélisme intrinsèque des circuits numériques. Ainsi, les différentes constructions du VHDL, à savoir :

Les process,
Les clauses de connexion,
Les assignations du type "flot de données",

sont toutes ramenées à un modèle de processus parallèles communicants.

Le but de la simulation logique d'un circuit numérique est de fournir l'état des signaux désirés de la description au cours du temps. La simulation d'une description VHDL revient à l'évaluation d'un ensemble de processus parallèles communiquants grâce aux signaux de leur interface. Or un ordinateur a un fonctionnement séquentiel, il est donc nécessaire de gérer la parallèlisation des processus, c'est le but des techniques de simulation.

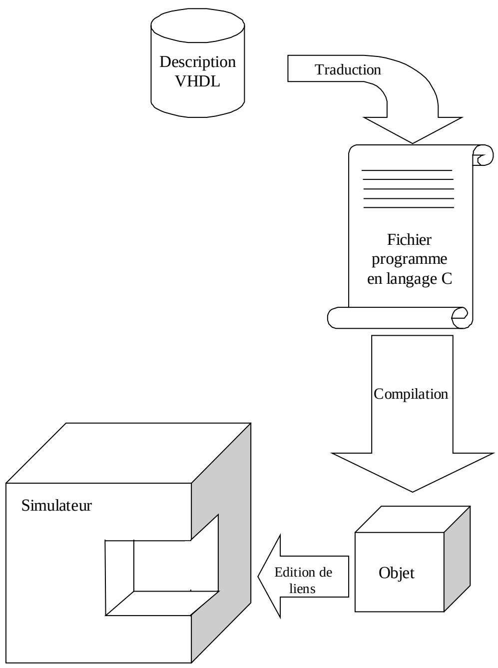
Il existe deux techniques de simulation : une technique compilée et une technique interprétée. Nous allons d'abord présenter ces deux techniques de simulations, puis nous étudierons les qualités et les défauts de chacune de ces techniques.

Nous présenterons ensuite une nouvelle technique de simulation que nous nous proposons de mettre en œuvre dans notre simulateur.

1) Etat de l’art

1.1) Techniques compilées

Dans une mise en œuvre compilée, un simulateur spécifique est construit pour chaque modèle à simuler.



Pour cela, la description du circuit est traduite en un langage directement compilable par l'ordinateur. Cette traduction fait appel à des fonctions ou procédures propres au noyau de simulation, elle doit être compilée et liée avec le noyau de simulation afin d'obtenir un programme exécutable.

Les mises en œuvre compilées de simulateur VHDL effectuées dans l’industrie traduisent généralement le VHDL en Langage C, puis le compilent et effectuent l’édition de lien avec le noyau de simulation. Le langage C est souvent choisi comme langage cible en raison de sa disponibilité sur diverses plates-formes, rendant ces programmes portables sur différentes architectures. De plus, les compilateurs C sont capables de fournir un code exécutable fortement optimisé.

Chapitre V

Techniques de simulation

4) Conclusion

La norme VHDL (IEEE 1076-1987) décrit dans son paragraphe 12.6, la technique d'exécution d'une description. Il est clairement spécifié que la simulation utilise la propagation de signaux et que le noyau du simulateur détecte des événements sur des signaux afin de réveiller et d'exécuter les processus dépendants de ces signaux.

Le noyau du simulateur doit posséder en permanence la valeur de tous les signaux de la description. Le temps de simulation géré par le noyau du simulateur est global (12.6.1), de plus chaque signal peut utiliser des attributs comme *stable, quiet, transaction* qui ne sont gérables qu'avec un algorithme event-driven ou de relaxation.

Le langage VHDL est sémantiquement très orienté vers l'utilisation de l'algorithme event-driven.

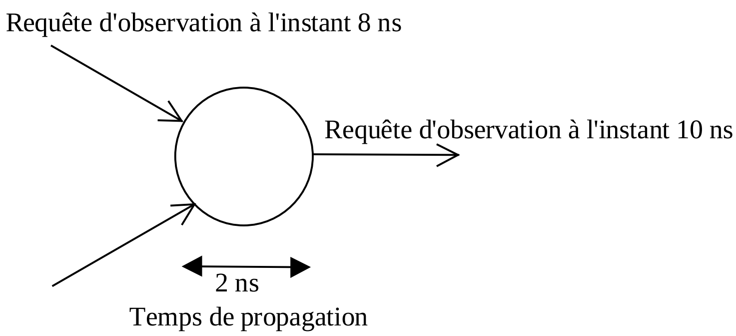
Or, une analyse des algorithmes de simulation montre clairement que l'algorithme event-driven réalise un compromis défendable entre la diminution des nombres d'évaluations et la complexité de la gestion du simulateur. Aussi, nous avons adopté cet algorithme de simulation pour la réalisation du prototype de notre simulateur.

3) Algorithme Demand-driven

L’algorithme de simulation *demand-driven* est l'algorithme entraînant le minimum d'évaluations de processus connu à ce jour [SMI87]. Au lieu de propager de façon aveugle des événements à travers le circuit. Il cherche uniquement à atteindre le résultat attendu sur les sorties observables.

L'algorithme demand-driven, ne cherche qu'à obtenir la valeur des signaux désirés. Il va ainsi, effectuer une rétropropagation en partant des sorties du circuit que l'on désire observer.

Le concept de base de l’algorithme *demand-driven* est simple : l'algorithme démarre par la requête d'observations de la valeur d'un signal (aussi appelé Demande de valeur de signal) à un instant donné. Cette requête est propagée vers les entrées du processus qui calcule la valeur de ce signal, en prenant soin de retrancher le temps de propagation à travers le process.



Cette opération est effectuée récursivement jusqu'à atteindre des signaux dont les valeurs sont connues aux instants indiqués par les requêtes. Alors commence une deuxième phase. Les valeurs sont propagées dans le réseau dans le sens normal jusqu'à atteindre le signal sur lequel on avait émis la requête initiale.

Une des différences fondamentales entre les algorithmes event-driven et demand-driven est l’absence d’échéancier durant l’exécution de l’algorithme demand-driven. Au lieu de cela, un événement sur un signal est attaché au signal lui-même. De la même façon, les stimuli d’entrée sont appliqués directement aux signaux d'entrée avec ***l'intervalle de validité temporelle*** pendant lequel cette valeur est valide.

L'algorithme demand-driven utilise la rétropropagation des dépendances à travers le circuit et le temps, afin de collecter les valeurs des signaux. De cette manière, il est possible d'éviter l'évaluation de processus qui ne contribuent pas à la valeur des signaux observés à l'instant requis.

Ainsi cet algorithme est celui qui demande le minimum d'évaluation de processus, malheureusement la récursivité de la rétropropagation des demandes est un mécanisme difficile à mettre en place. Pratiquement aucun simulateur logique n'utilise cet algorithme qui minimise le nombre de processus à évaluer, mais qui en même temps présente un accroissement du temps dédié à la gestion de la simulation.

Les simulateurs *event-driven* sont basés sur l'idée qu'il n'est pas nécessaire d'évaluer un processus tant que les valeurs de toutes ses entrées sont stables. Ainsi, ces simulations sont appelées "simulation dirigée par événement" ou *event-driven*.

Cet algorithme de simulation est présenté à la figure 4.3.

Les données nécessaires aux simulateurs event-driven sont : la description du circuit, les stimuli à appliquer aux entrées du circuit et la liste des signaux à observer.

Comme dans l'algorithme de relaxation, une initialisation du simulateur est nécessaire : tous les stimuli sont enregistrés dans l'échéancier. Une fois que cette tache est finie, le simulateur entre dans la boucle principale, et la simulation se déroule jusqu'à ce qu'il n'y ait plus dans l'échéancier de transactions enregistrées.

Au début de la boucle principale, l'échéancier est examiné afin de déterminer l'instant le plus proche pour lequel une transaction est enregistrée. A l'intérieur de la boucle principale et après l'étape ***incrémentation du temps*** qui gère le temps de simulation courant, il y a typiquement deux boucles : ***la boucle de mise à jour*** et ***la boucle d'évaluation.***

La boucle de mise à jour où chaque transaction prévue au temps de simulation courant (toujours représenté par une variable globale) est retirée de l'échéancier et où les signaux sont mis à jour. Un test permet de vérifier si chacune de ces transactions est un événement. Dans ce cas, les processus dépendants de ces événements sont placés dans ***la pile d'évaluation***, et l'événement est enregistré dans l'échéancier au temps de simulation courant incrémenté du temps de propagation du processus. Il faut noter que tout événement sur un signal invalide les événements prévus dans l'échéancier sur ce même signal à des temps ultérieurs. Cette boucle se déroule jusqu'à ce qu'il n'y ait plus d'événement prévu au temps courant de simulation.

La boucle d'évaluation est constituée de deux parties. La première, où un processus est dépilé de la pile d'évaluation. La seconde, où ce processus est évalué afin de déterminer les valeurs des signaux de sortie. Cette boucle se déroule jusqu'à ce que tous les processus présents dans la pile aient été évalués.

Le gain de temps provient donc ici, d'une diminution du nombre des processus à évaluer, la pile d'évaluation est chargée dynamiquement, selon les événements, et ne sont finalement évalués que les processus pouvant subir un changement d'état du fait d'un événement à leur entrée.

Néanmoins certains processus sont évalués plusieurs fois, en effet, on n'attend pas que les entrées soient stables afin d'évaluer le processus. Ainsi, un événement sur une entrée entraîne l'évaluation systématique du processus. Si un événement a lieu sur une autre entrée au même temps de simulation mais dans une autre itération de la boucle d'évaluation, ce processus sera réévalué.

Cet algorithme n'est donc pas celui qui entraîne le nombre minimum d'évaluation de processus.

Un tel ordonnancement est inapplicable dés que le graphe de dépendance entre les processus contient des cycles, ce qui est malheureusement le cas général.

2) Algorithme Event-driven

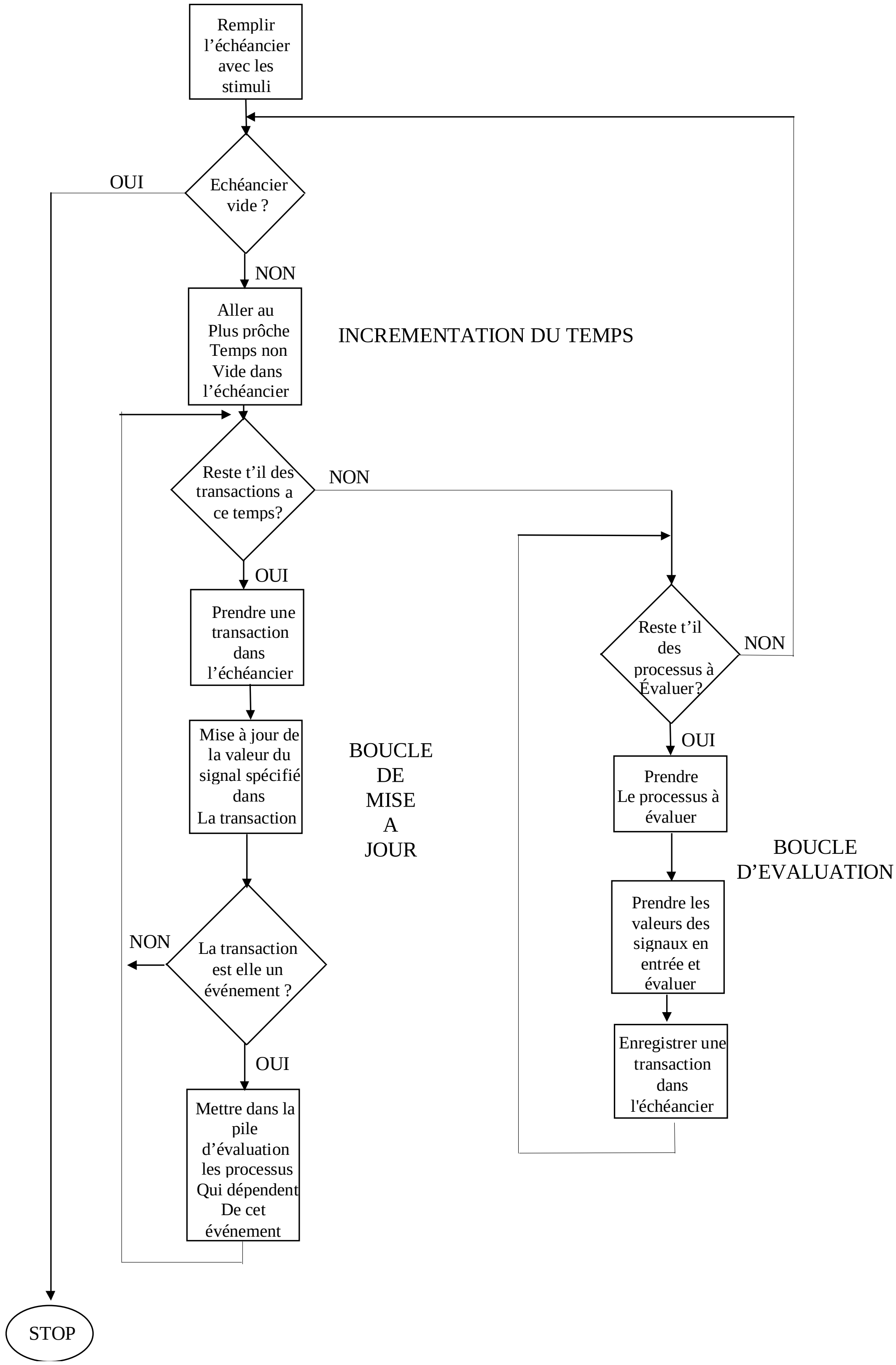


Figure 4.3. Simulation Event-Driven

typiquement trois sous-boucles : ***la boucle de mise à jour***, ***la boucle d'évaluation*** et ***la boucle de comparaison***.

La boucle de mise à jour, où les événements prévus au temps de simulation courant (une variable globale) sont retirés de l'échéancier et où les signaux sont mis à jour.

Puis tous les processus du circuit sont placés dans ***la pile d'évaluation*** afin d'être évalués. Cette phase initialise ***la boucle d'évaluation*** qui se déroule jusqu'à ce que tous les processus présents dans la pile aient été évalués.

La boucle de comparaison, vérifie si le circuit est devenu stable en vérifiant qu'aucun signal n'a changé de valeur par rapport à l'itération précédente. Si le circuit est stable, les événements sont enregistrés dans l'échéancier. Enfin, le temps de simulation courant est incrémenté et la boucle principale est reprise.

Si cet algorithme permet de simuler correctement le fonctionnement d'un circuit décrit de façon concurrente, il est largement sous-optimal. En effet à chaque itération, un grand nombre de processus sont évalués inutilement. Ceux ci sont les processus dont les entrées se sont stabilisées depuis plusieurs itérations. Leur évaluation produit fatalement le même résultat en sortie.

Ordonnancement

Une optimisation consiste donc à ordonner partiellement l'ensemble des processus naturellement concurrents qui représentent le circuit.

Si le graphe orienté de dépendance entre les processus est un graphe acyclique, on peut établir une relation d'ordre partielle entre les processus. En évaluant les processus dans cet ordre partiel, il est possible de réduire le nombre d'itérations.

Par exemple, considérons un circuit de trois portes :

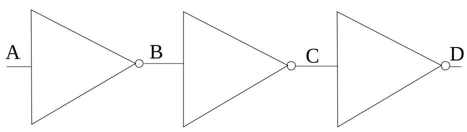


Fig. 4.2, Chaîne de trois inverseurs.

Dans l'exemple ci dessus, les trois équations logiques qui décrivent ce circuit, peuvent être évaluées une seule fois si l'ordre d'évaluation des expressions est le suivant :

B <- NOT A
C <- NOT B
D <- NOT C

L'ordonnancement nous permet donc d'évaluer ce circuit en une itération, là où l'algorithme classique de la relaxation aurait nécessité 4 itérations (3 itérations pour stabiliser le circuit et une itération pour déceler la stabilité).

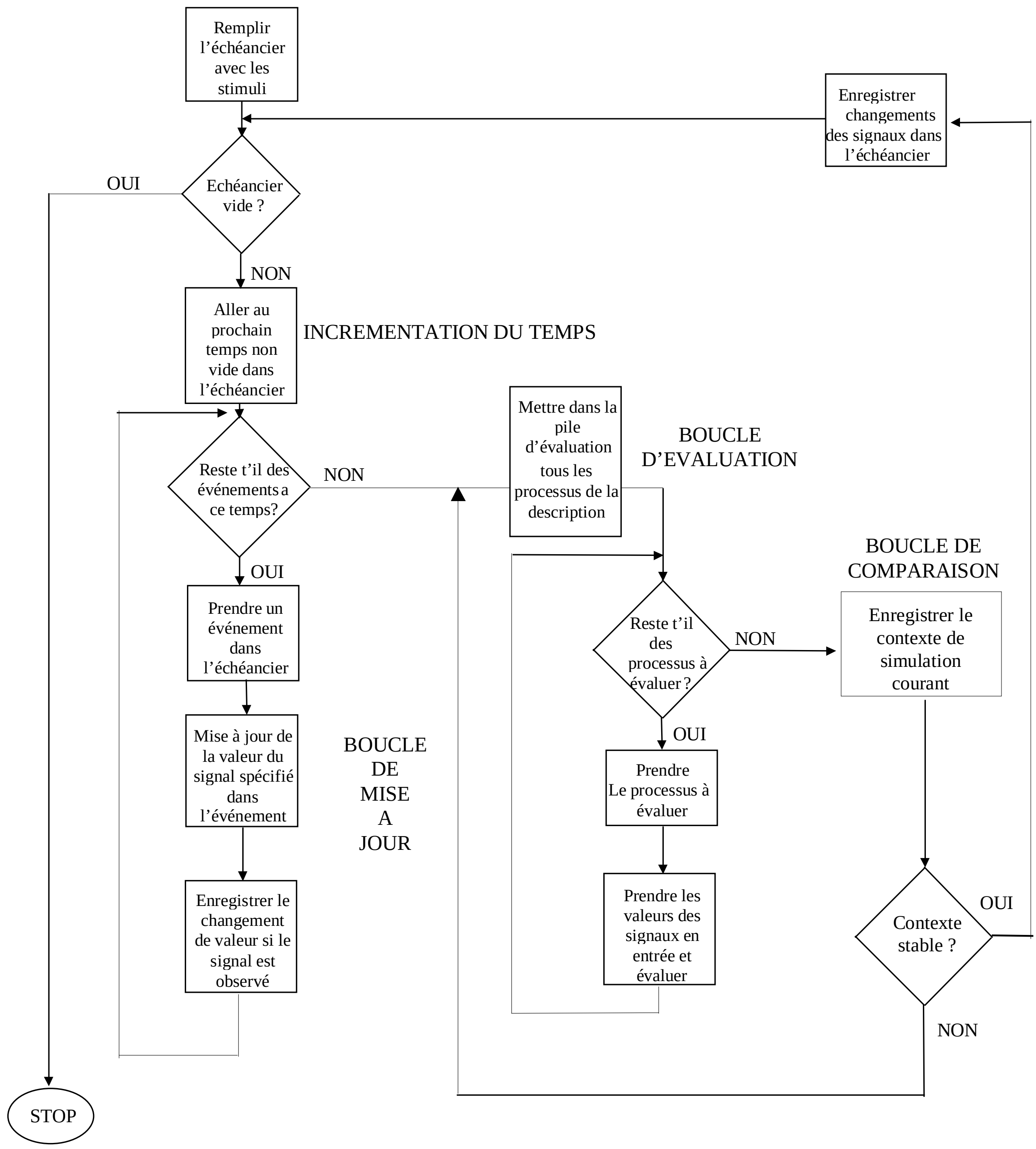


Fig. 4.1 : Algorithme de la relaxation

Les données nécessaires aux simulateurs à relaxation sont : la description du circuit, les stimuli à appliquer aux entrées du circuit et la liste des signaux à observer.

Avant de commencer la boucle principale de simulation, tous les stimuli sont enregistrés dans l'échéancier. Une fois que cette tache est finie, le simulateur entre dans la boucle principale, et la simulation se déroule jusqu'à ce qu'il n'y ait plus d'événements enregistrés dans l'échéancier.

Au début de la boucle principale, l'échéancier est examiné afin de déterminer l'instant le plus proche pour lequel un événement est enregistré. A l'intérieur de la boucle principale et après l'étape ***incrémentation du temps*** qui gère le temps de simulation courant, il y a

Comme nous l'avons vu, un système décrit en VHDL peut se ramener à un ensemble de processus communicant entre eux par des signaux.

Le problème de la simulation d'une telle description provient essentiellement du fait que les ordinateurs utilisés pour simuler ces circuits sont fondamentalement séquentiels. Pour réussir à simuler ces descriptions parallèles, il faut donc parvenir d'une manière ou d'une autre à séquentialiser la simulation.

Le but des algorithmes de simulation est donc de parvenir à exécuter une description parallèle sur une machine séquentielle en simulant le caractère parallèle de cette description.

Dans ce chapitre, nous nous attachons à décrire les algorithmes de simulation les plus répandus en mettant en évidence les avantages et les défauts de chaque méthode pour aboutir au choix que nous avons fait pour notre outil.

Terminologie

Pour clarifier la description des algorithmes qui va suivre, nous allons préciser quelques termes que nous allons utiliser par la suite.

Un ***signal*** connecte une sortie d'un processus à une ou plusieurs entrées d'un ou de plusieurs autres processus.

Les signaux ***observés***, sont les signaux dont la valeur est sauvegardée durant la simulation pour l'analyse ultérieure

Un ***événement*** a lieu quand un signal change de valeur. Il est défini par trois paramètres : un identificateur de signal, une nouvelle valeur à imposer sur le signal et le temps où cette valeur doit être imposée sur le signal.

Un ***stimulus*** est un événement d'un type particulier. Les stimuli permettent de modéliser le monde extérieur, l'environnement dans lequel fonctionne le système.

Un ***échéancier*** est un mécanisme logiciel permettant de stocker des événements, qui offre un système d'accès selon des critères temporels. C'est le mécanisme qui permet de gérer la chronologie des événements à l'intérieur du circuit.

1) Algorithme de la relaxation

L'algorithme de relaxation se base sur la recherche de la stabilité du circuit. Pour calculer l'état du système à un instant donné, on évalue itérativement tous les processus du circuit dans un ordre quelconque, jusqu'à ce que le système se stabilise

En effet, il est possible de démontrer par récurrence, qu'un système composé de n processus dépendant les uns des autres se stabilise au maximum après n itérations d'évaluation de l'ensemble des processus, à condition que le graphe de dépendance ne contienne pas de cycles.

Chapitre IV

Algorithmes de simulation

Terminologie

1) Algorithme de la relaxation
Ordonnancement

2) Algorithme Event-driven

3) Algorithme Demand-driven

4) Conclusion

composants utilisés dans le circuit, et, d'autre part, la manière dont ces composants sont interconnectés entre eux.

4.2.3) Description comportementale

Le circuit est modélisé par un ensemble d'assignations concurrentes décrivant un réseau Booléen.

Les éléments mémorisants sont identifiés par une déclaration explicite. Ces signaux sont déclarés comme des signaux résolus et gardés de type "*register*". L'assignation d'un élément mémorisant s'effectue à l'intérieur d'une instruction "*BLOCK*" associée à une condition de connexion/déconnexion (appelé condition de garde).

Les opérateurs Booléens acceptés sont les fonctions logiques préexistantes de VHDL : **NAND**, **AND**, **OR**, **OR**, **XOR**, **NOT**, **=** (égal), **/=** (différent) et la fonction **&** de concaténation. Ces opérateurs sont utilisés lors d'assignation de signaux. Toutes les formes d'assignation sont permises, assignation simple, assignation conditionnelle (de type IF..THEN..ELSE) ou assignation sélective (de type SWITCH..CASE).

5) Conclusion

Le choix du langage VHDL nous permet de simplifier le développement de notre simulateur. En effet, VHDL nous fournit un langage totalement spécifié à sémantique définie.

Nous avons du restreindre le VHDL à deux sous-ensembles structurel et comportemental, acceptables pour des outils de synthèse logique, de preuve formelle ou d'abstraction fonctionnelle, qui manipulent tous des réseaux Booléens. La grammaire définissant précisément ces deux sous-ensembles est décrite dans l'annexe 1 de ce manuscrit.

Ce sous-ensemble, obtenu en éliminant les constructions temporelles et procédurales, convient bien aux circuits numériques synchrones. Il reste exploitable par les concepteurs de circuit puisqu'il a été utilisé pour des circuits de forte complexité contenant plusieurs centaines de milliers de portes [GRE94][REB97][BAZ94].

4.2) Définition du sous-ensemble VHDL

Notre sous-ensemble accepte deux types d'architecture : l'architecture "structurelle" permet de décrire une "*net-list*" hiérarchique, l'architecture "flot de données" (*data-flow*) permet de décrire les réseaux Booléens qui constituent les éléments terminaux de la hiérarchie. VHDL permet de mixer les constructions structurelles et les assignations concurrentes dans le même fichier de description. Pour des raisons pédagogiques, nous décidons de séparer les deux types de description. Un même composant pourra avoir deux architectures, mais chacune des descriptions devra être faite dans un fichier séparé.

Ainsi il sera possible de décrire une même entité de deux façons différentes : une description structurelle (interconnexion de composant) et une description comportementale de niveau RTL n'utilisant que des assignations concurrentes (flot de données).

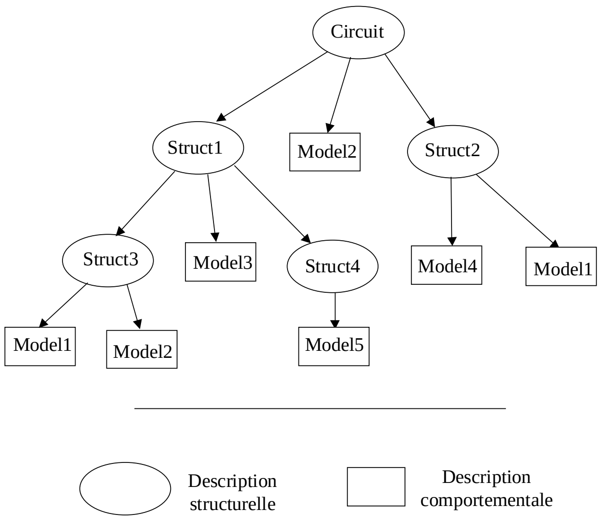


Figure 3.2, Description Hiérarchique d'un circuit

4.2.1) Type des signaux

VHDL est un langage permettant la définition de type de données. Pour la simulation, nous nous limiterons aux types de base bit et bit_vector.

Nous supportons les signaux multi-émetteurs comprenant un mécanisme de connexion/déconnexion. Ces signaux sont prédéfinis et au nombre de deux, il s'agit des bus construits à partir d'émetteurs trois états (*Tristate*) et des signaux préchargés réalisant la fonction OU câblé (*Wired OR*). Ces signaux ne diffèrent que par la fonction de résolution associée.

L'assignation d'un signal multi-émetteurs s'effectue à l'intérieur d'une instruction "*BLOCK*" associée à une condition de connexion/déconnexion (appelé condition de garde).

4.2.2) Description structurelle

Il s'agit d'une description de connectique, le circuit est décrit en terme des composants qui le constituent. Dans ce type de description, il faut donc, d'une part, spécifier les

Une expression Booléenne peut être vue comme une représentation textuelle d'une fonction Booléenne.

Soit F une fonction Booléenne et E une expression Booléenne alors F=E est interprété comme "la fonction Booléenne F peut être représentée par l'expression Booléenne E".

On étend l'ensemble des opérateurs Booléens à $\left\{.,+,\overline{\ },\oplus\right\}$, $\oplus$ est l'opérateur Booléen XOR (OU exclusif). On définit l'ensemble des expressions Booléennes à travers le système formel suivant [CHI67] :

1) 0 et 1 sont des expressions Booléennes.
2) Toute variable Booléenne x est une expression Booléenne.
3) Si A est une expression Booléenne, alors $\overline{A}$ en est également une.

Si A et B sont deux expressions Booléennes, alors $A+B$, $A.B$, $A \oplus B$ le sont également.

4) Une expression Booléenne ne peut être obtenue que par les règles 1, 2, 3 et 4 appliquées un nombre fini de fois.

Réseau Booléen

Un réseau Booléen est un graphe orienté dans lequel chaque nœud est associé à un nom de variable Yi et à une fonction Booléenne Fi représenté par une expression Booléenne Ei.

Un réseau Booléen permet de représenter un circuit combinatoire. On peut étendre cette définition pour prendre en compte les registres quand ceux-ci sont déclarés de façon explicite. L'exemple suivant donne le code VHDL en flot de données d'un petit circuit contenant un registre reg et son schéma équivalent en terme de réseau Booléen :

```
-- Interface du circuit
ENTITY circuit IS
  PORT ( a, b, c : IN BIT;
        d, e, clk: IN BIT;
         x, y, z : OUT BIT);
END circuit;

- - Comportement du circuit
ARCHITECTURE data_flow
OF circuit IS

SIGNAL u, v, w : bit;
SIGNAL reg     : reg_bit

BEGIN
v <= a or e;
w <= v xor not(d);
u <= b and c;

L0 : BLOCK ((clk = '0'))
  BEGIN
    reg <= GUARDED not u;
  END BLOCK;

z <= w and u;
x <= not reg;
y <= a xor u;
END;
```

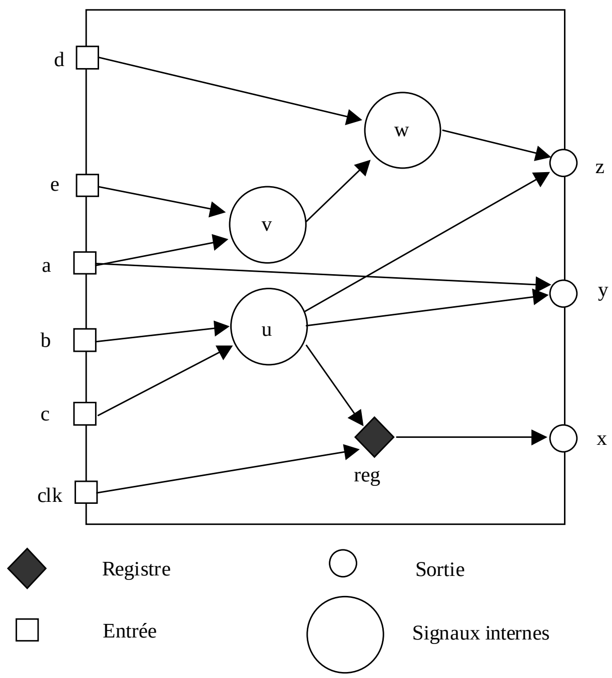


Fig. 3.1, Réseau Booléen de la description VHDL

circuit réalisé. Il est donc une alternative à la co-simulation. A l'heure actuelle, seuls les outils de preuve travaillant au niveau Booléen sont réellement utilisables. Il doit exister une correspondance bijective entre les registres instanciés dans les deux descriptions. La preuve formelle consiste à comparer deux réseaux Booléens ayant les mêmes entrées et les mêmes sorties.

On voit que les outils de la chaîne de CAO Alliance manipulant des descriptions comportementales tels que la synthèse logique, la preuve Booléenne et l'abstraction fonctionnelle ne peuvent pas traiter les clauses procédurales du VHDL. De plus, les délais ne sont pas utilisables par ces outils, les modèles qu'ils utilisent sont des modèles de type "*zero-delay*".

Notre simulateur utilise le même sous-ensemble VHDL que ces outils. Notre sous-ensemble du langage VHDL est donc le VHDL amputé des clauses procédurales et temporelles, nous garderons les clauses structurelles permettant de décrire une structure hiérarchique multi-niveaux et les clauses de type assignations concurrentes.

Cette restriction nous permet de garantir l'interopérabilité des fichiers de description utilisant notre sous-ensemble avec les outils de synthèse logique, d'abstraction fonctionnelle et de preuve Booléenne.

4) Le sous-ensemble

Nous cherchons à représenter des réseaux Booléens. C'est pourquoi, nous allons d'abord définir la terminologie que nous utilisons.

4.1) Algèbre Booléenne

L'ensemble B={0,1} muni des opérations . (ET), + (OU) et $\bar{\ }$ (NON) forment une algèbre de Boole [BOO97].

Variable Booléenne
Une variable Booléenne est une variable à valeur dans B.
Notation : si x est une variable, son inverse est noté $\bar{x}$

Fonction Booléenne
Une fonction Booléenne est une fonction discrète de variables Booléennes (x_1, ..., x_n) définies dans B^n et prenant sa valeur dans B.

Support d'une fonction Booléenne
On appelle support d'une fonction Booléenne, l'ensemble des variables Booléennes dont la fonction dépend.

Expressions Booléennes
Une expression Booléenne est la représentation d'une fonction Booléenne qui contient des opérateurs Booléens, de variables Booléennes, des constantes (0 et 1) et un nombre quelconque de niveau de parenthèses.

Exemple : $(a+b+c).(c+d+0)+a.\bar{e}.1$ est une expression Booléenne.

- Les clauses d'assignations concurrentes de signaux (description "flot de données").
- Les clauses d'instanciation de composants (description structurelle).

Ces clauses permettent de décrire le parallélisme d'exécution à l'intérieur d'un circuit. La sémantique de simulation de VHDL considère ces trois types de clauses comme des processus parallèles.

Ces processus peuvent être activés par les événements se produisant sur leur interface. VHDL permet de mixer ces trois types de clauses (procédural, structurel, flot de données) à l'intérieur d'une même description.

Les processus de la description communiquent entre eux par des signaux. La phase d'élaboration transforme une description VHDL quelconque en un ensemble homogène de processus parallèles communicants.

Afin de pouvoir répondre aux besoins du plus grand nombre d'utilisateurs, VHDL a été conçu, dés l'origine comme un langage ouvert. Il permet notamment de redéfinir l'ensemble des valeurs attribuées à un signal ainsi que les fonctions et opérateurs agissant sur ces signaux [LOU88] [SHA86]. VHDL fournit un véritable langage de programmation, très proche en philosophie d'ADA.

VHDL, toutefois, possède les défauts de ses qualités. En effet, la richesse du langage dépasse largement nos besoins. Cette richesse est un handicap pour le développement d'un compilateur. Il semble donc raisonnable, dans ce cas de figure, de nous restreindre à un sous-ensemble adapté aux circuits numériques synchrones.

3) Comment restreindre VHDL ?

La spécification du sous-ensemble VHDL à adopter a été effectuée grâce à l'analyse des besoins des autres outils de la chaîne de CAO Alliance, utilisateurs potentiels de ce sous-ensemble.

• **Synthèse Logique** : Tous les outils de synthèse logique génèrent une interconnexion de portes logiques à partir d'une description comportementale qui est en pratique un réseau Booléen. Il faut pouvoir clairement identifier les différents types d'éléments mémorisants à synthétiser. Remarquons ici que la représentation la plus directe d'un réseau Booléen en VHDL est une description de type "flot de données" où chaque signal possède une assignation unique.

• **Abstraction Fonctionnelle** : Ce type d'outil regénère le comportement d'un circuit en partant de sa "*net-list*" au niveau transistors. La première étape de l'abstraction fonctionnelle consiste à reconstruire une "*net-list*" de portes orientées à partir de la "*net-list*" de transistors. La seconde consiste à attacher à chaque porte une expression Booléenne correspondant à la fonction logique réalisée. On obtient donc à la fin un réseau Booléen qu'il est possible de décrire en VHDL "flot de données".

• **Preuve Booléenne** : Cet outil permet de vérifier que deux descriptions comportementales possèdent la même fonctionnalité. Il permet de comparer la description comportementale de départ servant à la spécification du circuit et la description comportementale extraite du

Lors de la conception d'un simulateur logique, la première étape et sans doute la plus essentielle, est la spécification du langage de description reconnu.

Nous étudierons dans un premier temps, l'utilisation des langages de description de matériels. Puis, nous présenterons les raisons qui nous ont fait choisir le langage VHDL, nous définirons les concepts Booléens que nous manipulerons, enfin nous présenterons le sous-ensemble de ce langage que nous avons été amenés à spécifier.

1) Utilité

Dans les années 60, le développement des premiers circuits intégrés faisait principalement appel à la simulation électrique. Le besoin en simulation logique est apparu avec l'augmentation du taux d'intégration des transistors. Avec cette augmentation, la vérification de circuits comportant un grand nombre de transistors au moyen d'un simulateur électrique devient irréalisable, aussi bien à cause de la taille de la mémoire nécessaire, que du temps total de simulation. Il a donc fallu utiliser des simulateurs de plus haut niveau, adaptés aux circuits numériques de haute complexité.

De ce besoin de simulation logique, sont nés les langages de description de matériels appelés encore HDL (*Hardware Description Language*). En effet, les langages séquentiels classiques sont mal adaptés à la description du comportement intrinsèquement parallèle d'un système matériel. De plus, certaines caractéristiques des circuits telles que les délais de propagation sont difficiles à exprimer dans un langage de programmation classique.

La description d'un circuit à l'aide d'un langage spécifique poursuit deux buts. Le premier et le plus évident, est de disposer d'une saisie informatique du circuit, pour qu'elle puisse être traitée par des outils de CAO. Mais, une description de circuit est également un moyen de documentation, et les concepteurs utilisent cette description comme un support à leurs réflexions et discussions. Les langages de description de matériels doivent donc répondre à deux exigences qui peuvent être antagonistes. Ils doivent avoir une sémantique et une syntaxe appréhendables par l'homme et manipulables par des outils de CAO.

2) Particularités de VHDL

VHDL [IEEE88][AIR90] est un langage de description de circuits, développé par IBM, Intermetrics, ATT,... sous l’impulsion du DOD (*Department Of Defense, USA*). Ce langage s'est imposé comme une norme de fait après l'annonce faite par le DOD de ne plus acheter que des composants accompagnés d'une description en langage VHDL. VHDL a été déposé dans le domaine public, par le biais de l’IEEE (*Institute of Electricals and Electronic Engineers*), sous la référence 1076-1987.

Ce langage vise à fournir un moyen pour la description des circuits VLSI, commun à tous les concepteurs et intégrateurs de circuits intégrés [DEW84]. Les différents niveaux d'abstraction possibles du VHDL le rendent utilisable par les différentes catégories d'utilisateurs potentiels.

En effet, VHDL accepte trois grands types de clauses :

- Les clauses procédurales (utilisation de process, description séquentielle).

Chapitre III

Le langage VHDL

gamme. En effet, le coût des stations de travail professionnelles et surtout des logiciels professionnels de CAO, est tellement important, qu'il est difficilement compatible avec un enseignement de masse.

Elle comporte également des outils avancés issus de programmes de recherche, afin d'offrir aux concepteurs de circuits des outils évolués (abstraction fonctionnelle, preuve formelle, etc.), non encore disponibles dans les chaînes commerciales.

Enfin, la chaîne Alliance fournit une plate-forme logicielle permettant d'expérimenter de nouveaux algorithmes ou de nouveaux outils de CAO.

Développer un outil de CAO à l'intérieur d'une chaîne existante présente des avantages, mais également des servitudes. Le développement du simulateur a été fortement influencé par la contrainte d'intégration du simulateur dans la chaîne Alliance. Les structures de données représentant les informations structurelles (*net-list*) d'un circuit étaient déjà définies, il fallait s'adapter à ces structures génériques, et ne pas céder à la tentation d'inventer nos propres structures de données développées sur mesure pour notre outil.

De plus, les structures de données manquantes (vues comportementales, représentation des stimuli) ont été développées dans la même philosophie et suivant une optique programmation objet. Ces structures de données doivent être suffisamment génériques pour pouvoir être utilisés par d'autres outils (synthèse logique, preuve formelle, etc.).

Enfin, la portabilité impose la possibilité de l'utilisation du simulateur sur des machines bas de gamme. C'est à dire avec des processeurs peu puissants, et peu de mémoire.

5) Conclusion

Nos travaux ont pour but d'expérimenter la simulation logique de circuits numériques synchrones par l'évaluation directe d'un réseau Booléen représentant le circuit à simuler. L'utilisation de réseaux booléens afin de représenter les circuits permet de ramener l'évaluation de la description à un parcours de graphe.

Comme le langage VHDL contient des constructions qui ne peuvent pas se ramener à un réseau Booléen, nous allons définir un sous-ensemble du langage VHDL.

Le principal objectif de ce travail est donc d'évaluer les performances d'un tel modèle de simulation.

propriétaire de Cadence, sa conception est antérieure à celle de VHDL. Mais devant le danger que représentait la diffusion importante du VHDL, Cadence a également déposé son langage dans le domaine public. On peut estimer que Verilog et VHDL sont utilisés dans le monde de manière équitable, VHDL étant préféré en Europe et Verilog aux USA. Pour notre part, nous avons choisi d'utiliser VHDL.

La principale utilité d'un langage de description de matériel est de fournir des constructions permettant de décrire le parallélisme d'un système matériel constitué d'un grand nombre de composants fonctionnant en parallèle.

Le langage VHDL fournit principalement trois constructions pour décrire ce parallélisme :

- Instanciation de composants permettant des descriptions structurelles.
- Assignations concurrentes de signaux permettant des descriptions de type "flot de données".
- Définition de processus s'exécutant en parallèle avec une synchronisation par signaux.

Si la sémantique de simulation du langage VHDL est parfaitement définie pour toutes les constructions du langage, d'autres outils de CAO tels que les outils de synthèse logique ou de preuve Booléenne en particulier, n'acceptent qu'un sous-ensemble du langage VHDL. Afin de garantir l'interopérabilité entre les divers outils de CAO, nous allons donc réduire le langage VHDL à un sous-ensemble utilisable par l'ensemble des outils travaillant au niveau Booléen.

3) Simulation logique ou logico-temporelle

Nous ne nous intéresserons qu'à la seule simulation logique en écartant tout l'aspect temporel. En effet, la méthodologie utilisée dans notre laboratoire d'accueil, ainsi que par un nombre croissant d'industriels, sépare la vérification de fonctionnalité logique, de la vérification temporelle. Nous considérons que la simulation logico-temporelle ne permet plus aujourd'hui de calculer la fréquence de fonctionnement maximale d'un circuit contenant plusieurs millions de transistors, il faut utiliser des méthodes formelles et un outil d'analyse temporelle statique [HAJ92]. Un tel découplage ne s'applique évidemment qu'aux circuits numériques synchrones.

4) Contexte : la chaîne Alliance

Alliance [GRE93] est une chaîne d'outils de conception pour les circuits intégrés. Cette chaîne a été développée au département ASIM (Architecture des Systèmes Intégrés et Micro-électronique) du Laboratoire LIP6 (Laboratoire d'Informatique de Paris 6). Elle vise la conception des circuits complexes pour la recherche ainsi que la conception de circuits à but pédagogique. De plus, cette chaîne se propose de couvrir toutes les étapes de la conception suivant une méthode de conception descendante, depuis la spécification fonctionnelle jusqu'au dessin des masques.

Le premier objectif d'Alliance est de pouvoir fournir gratuitement aux chercheurs et aux universitaires une chaîne de CAO complète, utilisable sur des plates-formes Unix bas de

Ces chiffres sont confirmés par M. TABUSSE [TAB94] qui indique un ratio de 70% pour la vérification d'un ASIC et 30% pour la définition de son schéma.

1) Objectif de la thèse

Nos travaux ont pour objectif d'explorer une nouvelle technique de simulation logique pour des circuits intégrés numériques synchrones de grande complexité. Dans le cadre de cette étude et afin d'évaluer la faisabilité et la pertinence de cette technique, nous avons conçu et développé un outil de simulation logique appelé Asimut.

Tous les simulateurs logiques développés jusqu'à présent, travaillent selon deux axes : ils utilisent, soit une technique interprétée, soit une technique compilée. La technique interprétée consiste à traduire la description en un pseudo-code qui sera exécuté sur une machine virtuelle. La technique compilée consiste, en général, à traduire la description en un programme séquentiel utilisant un langage de haut niveau tel que le langage C. Ce programme est compilé et lié avec un noyau de simulateur pour obtenir un exécutable.

La technique interprétée a pour principal défaut d'être lente, mais facile à développer et à mettre au point. La technique compilée est plus rapide à l'exécution, mais le moindre changement dans une description impose de recompiler l'ensemble de la description afin d'obtenir un nouvel exécutable.

Nous présentons dans cette thèse une méthode de simulation fondée sur l'évaluation directe des expressions Booléennes représentées en mémoire par différents types de graphes, tels que les arbres binaires (ABL) ou les graphes de décision binaire (*BDD*).

Cette approche est bien adaptée pour des circuits décrits comme des réseaux de portes ou comme des réseaux Booléens.

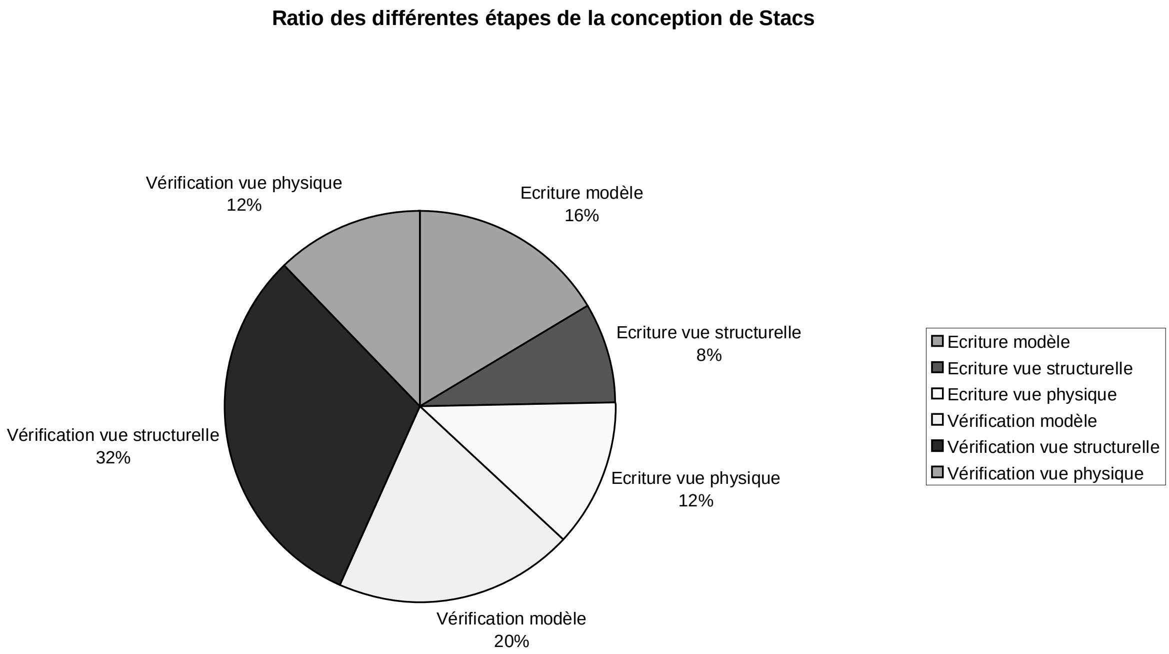
Les descriptions visées sont les modèles VHDL de type flot de données "*data-flow*" ne contenant que des assignations concurrentes où les signaux sont des "bits", et où on dispose d'une déclaration explicite des registres internes. Ce type de description représente en fait un réseau Booléen.

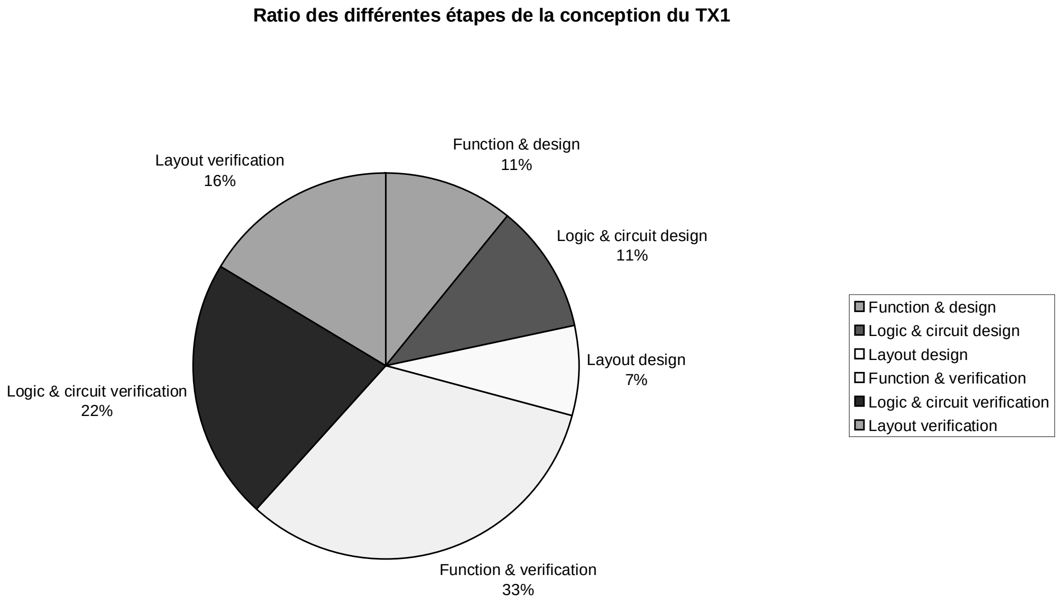
Ce travail a donc pour objectif la mise en place d'une plate-forme logicielle, afin d'expérimenter la pertinence de l'utilisation de graphes de représentation des expressions Booléennes dans le cadre de la simulation logique.

2) Les langages de descriptions de matériels d'aujourd'hui

La description de circuits et systèmes électroniques exige un langage spécifique. En effet, les langages de programmation séquentiels ne se prêtent pas à une telle utilisation.

En pratique, les langages utilisés pour la simulation logique se limitent à deux : Verilog [THO91] et VHDL [IEEE88]. VHDL est un langage développé sur l'initiative du DOD (Department Of Defense, le ministère de la défense des Etats-Unis d'Amérique) par trois compagnies américaines : IBM, Intermetrics, AT&T. Dès sa conception, ce langage a été déposé dans le domaine public et normalisé par IEEE. Verilog est à l'origine un langage





En effet, sur les deux exemples ci-dessus : le Toshiba TX1, processeur 32 bits, 460.000 transistors [USA89] et le Stacs, processeur VLIW, 875.000 transistors [LUC95], les étapes de vérification représente respectivement 71% et 64% du temps de conception.

Définition [SIM98] : La *simulation* est l'action d'utiliser un ordinateur pour imiter le comportement d'événements, d'activités ou de processus réels ou imaginaires.

Ce monde réel ou imaginaire est habituellement appelé *système*.

Des hypothèses sont faites sur ce système et des relations de cause à effet traduisent ces hypothèses, ceci constitue un *modèle* qui est supposé décrire le comportement du *système*.

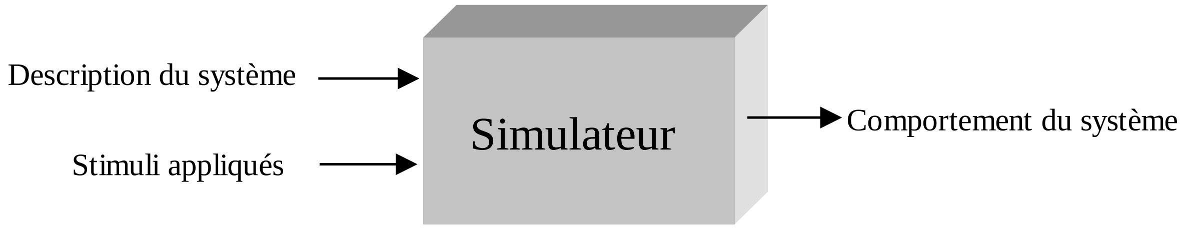


Fig. 1.1 Schéma général d'un simulateur

Un *simulateur* est un programme capable d'imiter le comportement d'un système à partir de sa description (son modèle) et des stimuli qui lui sont appliqués. Dans le cas qui nous intéresse, le système étudié est un circuit électronique.

Les concepteurs de circuits utilisent les simulateurs afin d'obtenir plusieurs types d'informations, portant sur le comportement logique du circuit, ainsi que sur sa consommation électrique et ses temps de propagation.

Il existe plusieurs types de simulateurs qui permettent d'obtenir ces informations. Les deux principaux types sont les simulateurs analogiques et les simulateurs logiques.

Les simulateurs analogiques (aussi appelés simulateurs électriques) sont utilisés pour obtenir des informations électriques sur le circuit, tel que temps de propagation, niveaux électriques ou forme des signaux. Il s'agit d'une simulation en temps continu où l'on calcule à tout instant les valeurs de tous les signaux. Ils sont cependant, très gourmands en temps de calcul et deviennent vite inexploitables pour des circuits dépassant quelques milliers de transistors.

Les simulateurs logiques sont eux utilisés pour des circuits ou systèmes numériques où les signaux sont de type Booléen. Si, de plus, le système considéré est synchrone, on peut discrétiser le temps en négligeant les temps de propagation à travers les portes logiques. Dans ce type de simulateur, on ne manie plus de courant ou de tension mais des signaux et des opérations Booléennes. Cette simplification dans la description du circuit s'accompagne d'une amélioration significative du temps d'exécution. Typiquement, un simulateur logique est 1000 fois plus rapide que le meilleur des simulateurs électriques.

Pour les circuits numériques synchrones, la simulation n'est pas la seule solution. Une autre solution existe : l'émulation. L'émulation consiste en la constitution d'une maquette matérielle du composant. Cependant un émulateur matériel constitue une solution efficace mais coûteuse. Nous ne nous intéresserons ici qu'à la seule simulation.

De manière générale, la simulation constitue l'étape la plus longue au cours d'un processus de conception de circuit.

Chapitre II

Problématique et contexte

La vérification fonctionnelle constitue une étape incontournable dans le processus de conception d'un circuit VLSI.

Actuellement, la majeure partie de la vérification de la fonctionnalité d'un circuit numérique synchrone s'effectue par simulation logique. Il y a un besoin permanent d'adapter les outils de simulation à l'évolution de la technologie.

Ainsi, le niveau d'intégration, toujours croissant, des circuits, constitue le défi majeur que les outils de simulation doivent continuellement relever pour permettre la validation de circuits de plus en plus complexes.

Ce manuscrit présente un simulateur logique, acceptant un sous-ensemble du langage VHDL adapté aux systèmes numériques synchrones.

La technique de simulation proposée dans cette thèse, s'appuie sur l'évaluation directe des expressions Booléennes représentées par des graphes.

Nous présentons la finalité de la thèse ainsi que le contexte de nos travaux dans le chapitre II. Nous présentons également la chaîne de CAO Alliance à laquelle est intégré notre outil ainsi que les contraintes de développement de notre simulateur.

Le chapitre III, présente la sémantique de simulation du langage VHDL qui est le langage de description adopté. Nous justifions le choix d'un sous-ensemble de ce langage.

Dans le chapitre IV, nous étudions les différents algorithmes de simulation existants, cette étude nous permet de choisir l'algorithme qui se prête le mieux à nos besoins.

Le chapitre V, nous permet de comparer les deux techniques usuelles de simulation : technique interprétée et technique compilée. Nous proposons une nouvelle technique de simulation qui utilise des structures de graphes construites statiquement avant la simulation.

Dans le chapitre VI, nous présentons les deux techniques de représentation des expressions Booléennes que nous avons utilisées, une première technique utilisant des arbres binaires (ABL), et une deuxième technique utilisant des diagrammes de décision binaires (BDD).

Le chapitre VII, expose l'architecture du simulateur Asimut. Nous présentons également les structures de données du simulateur qui nous permettront d'expliquer pas à pas, le déroulement d'une simulation sur un exemple.

Le chapitre VIII, présente des résultats de simulation et compare les performances de notre outil aux simulateurs de l'industrie proposés par Synopsys et Cadence, selon les critères de l'occupation mémoire et du temps d'exécution.

Enfin, dans le chapitre IX nous présentons nos conclusions et les perspectives de ce travail de recherche.

Chapitre I

Introduction

PLAN

Abstract

We present in this thesis, a simulation method for integrated circuits based on direct evaluation of logical expressions represented by graphs (BDD and Lisp Like Trees). The circuits are described with data structures.

We use a VHDL subset that excludes processes and timing information. This subset has been defined in order to be accepted by the tools of the Alliance VLSI CAD System that handle behavioural information (logic synthesis, formal proof, functional abstraction).

This subset has been used with success in research projects dealing with high complexity circuits.

The event-driven simulation technique has been used in an Alliance tool prototype called Asimut. The result of the comparison of this prototype versus industrials simulators (Cadence and Synopsys) shows than we obtain acceptables performances.

This prototype allowed us to adjust a software patform aimed at helping the development of tools that need an event-driven simulation kernel.

Keywords

Logical Simulation

VHDL

Graphs

Booleans Networks

Alliance

Event-driven

Résumé

Nous présentons dans ce manuscrit une méthode de simulation de circuits intégrés basée sur l'évaluation directe d'expressions logiques représentées par des graphes (BDD et ABL). La description des circuits est représenté à l'aide de structures de données.

Nous utilisons un sous-ensemble du langage VHDL excluant les *process* et les informations temporelles. Ce sous-ensemble a été défini de façon à être accepté par l'ensemble des outils de la chaîne de CAO pour VLSI Alliance manipulant des informations comportementales (synthèse logique, preuve formelle, abstraction fonctionnelle).

Ce sous-ensemble a été utilisé avec succès lors du développement de circuits de recherche de grande complexité.

La technique de simulation à pilotage événementiel (event-driven) a été utilisée lors de la mise en œuvre de cette méthode dans un prototype appelé Asimut, qui fait partie de la chaîne Alliance. Le résultat de la comparaison de ce prototype contre les simulateurs du commerce (Cadence et Synopsys) montre que l'on atteint des performances acceptables.

Ce prototype a permis de mettre au point une plate-forme logicielle facilitant le développement d'outils nécessitant un noyau de simulation à propagation d'événements.

Mots clés

Simulation logique

réseaux Booléens

VHDL

Alliance

Graphes

Propagation d'événement

Remerciements,

Je tiens à remercier tout particulièrement mon directeur de thèse Pirouz BAZARGAN SABET, qui par son amitié, son infinie patience et ses nombreuses réflexions a su m'aider à faire aboutir mes travaux et à concrétiser ce manuscrit.

Je remercie le Professeur Alain GREINER de m'avoir accueilli au sein de son équipe de recherche.

Je remercie les membres du jury d'avoir su se montrer disponible jusqu'à la date de soutenance.

Enfin je remercie les membres de l'équipe ASIM qui ont su créer une atmosphère de travail agréable et conviviale favorisant le déroulement de mes travaux :

Tout d'abord, le trio infernal :

Nizar	ABDALLAH
Gilles-Eric	DESCAMPS
Julien	DUNOYER

Sans eux, mes moments de doute et de désespoir inhérent à toute rédaction de thèse auraient été plus dur à surmonter.

Sans oublier :

Alexandre	FENYO
Olivier	FLORENT
Denis	HOMMAIS
Payam	KIANI
Luis	LUCAS
Frédéric	PETROT
Vincent	REBALDI
Pascal	REMY
Laurent	VUILLEMIN
Franck	WAJSBURT

Qu'ils soient ici remerciés de leur amitié, de leur soutien et de leur disponibilité.

Enfin, je remercie ma famille qui a du me supporter lors de la rédaction de ce manuscrit, eux aussi ont joués un rôle important dans l'aboutissement de ces années.

A mon père et à ma mère, avec toute mon affection

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

Spécialité : **INFORMATIQUE**

Présentée par :

VUONG Huu Nghia

Pour obtenir le titre de :

DOCTEUR DE L'UNIVERSITÉ PARIS VI

Sujet de la thèse :

Une nouvelle méthode de simulation par évaluation directe des expressions logiques représentées par des graphes : application à des circuits modélisés par un sous-ensemble du langage VHDL.

Soutenue le : 9 Décembre 1997

devant le jury composé de :

M. François	ANCEAU	Rapporteur
M. Pirouz	BAZARGAN SABET	Directeur de thèse
Mme Dominique	BORRIONE	Président du Jury
M. Alain	GREINER	Examinateur
M. Yannick	HERVE	Examinateur
Mme Anne Marie	TRULLEMANS	Rapporteur