Génération Automatique de Contrôleur Reprogrammable dans un Environnement de Synthèse de Haut Niveau

M. BENMOHAMMED [1,2],
P. KISSION [2] and A. A. JERRAYA [2].

[1] Laboratoire SDA, Institut d'Informatique, Université de SBA, 22000 Sidi Bel-Abbes, ALGERIE.
[2] Laboratoire TIMA/CNRS-INPG , 46 Av. Félix Viallet, 38031 Grenoble Cedex, FRANCE.

Mots clés: Synthèse Architecturale, Synthèse de haut niveau (HLS: High-Level Synthesis), Séquenceur, ASIP, ASIC, Contrôleur, ROM, VHDL.

RESUME

Ce papier traite de la synthèse automatique d'un contrôleur reprogrammable dans un environnement de synthèse de haut niveau. Partant d'une spécification comportementale de haut niveau décrivant la fonction du circuit, la synthèse de haut niveau fournit une architecture composée d'un chemin de données et d'un contrôleur. L'architecture générée est flexible du fait de l'utilisation d'un contrôleur à base d'une ROM et d'un séquenceur figé. L'avantage d'utiliser un contrôleur reprogrammable est de permettre: une réutilisation du chemin de données, une réduction du temps de conception et de profiter de la nature du matériel (rapidité). Une comparaison sera faite entre les trois solutions offertes actuellement au concepteurs de processeurs: processeur standard (du marché), processeur câblé (ASIC) ou ASIP.

ABSTRACT

In this paper we discuss the generation of reprogrammable controller within a high-level synthesis environment. Starting from a high-level behavioral specification (in algorithmic form) describing the circuit's function, the high-level synthesis provides an architecture composed of a data-path and a controller. The generated architecture is flexible due to the use of a controller based on a ROM (reprogrammable). The advantage of using a reprogrammable controller is to permit a reuse of the data-path, a reduction of the design time and to take advantage of the hardware nature. A trade-off between three solutions is done: Standard processor, ASIC or ASIP.

---

1. Introduction

Les applications programmables sont de plus en plus étudiées de nos jours, et se dévoilent d'un très grand intérêt. Elles émergent comme un nouveau style de conception dans les circuits intégrés ASIP [1] [19]. Les principales raisons de cette tendance sont:

- **flexibilité:** la programmabilité réduit l'impact des risques d'erreurs, en conception, les changements de dernière minute, ...
- **réutilisation:** il est plus facile d'adapter une architecture programmable que câblée,
- **réduction de la complexité:** la présence de blocs programmables peut simplifier énormément la conception de systèmes hétérogènes sur un circuit [1]. En plus, le traitement de signaux complexes est mieux géré en logiciel qu'en matériel.

Actuellement les différentes étapes du processus de conception automatique d'un circuit, de la spécification comportementale au "layout", s'emboîtent correctement. Cependant la synthèse d'un circuit reste une tâche complexe et longue. Un circuit est généralement composé d'une partie contrôle et d'une partie opérative au niveau architectural. Toute nécessité de modification de l'algorithme due à une erreur de spécification ou tout simplement à une mise à jour se traduit par la re-synthèse complète du circuit à partir des spécifications comportementales.

Les outils de synthèse actuels peuvent être catalogués comme suit:

- outils de synthèse de haut niveau qui [4] [11], à partir d'une description comportementale, génère une architecture (partie opérative, partie contrôle et une configuration du circuit global): ce sont des outils généralement destinés à la conception de processeurs dédiés câblés (ASIC),
- outils de génération de codes (voir figure 1: GC) [1] [9] qui, à partir d'une architecture cible et d'une description comportementale, fournissent le code correspondant: ce sont des outils destinés au processeurs dédiés programmables (ASIP).

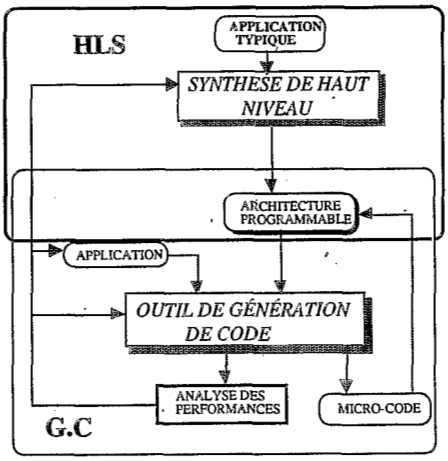

Figure 1: Conception flexible

Ce que l'on cherche à obtenir c'est la génération d'un chemin de données et un code correspondant pour la partie opérative. Pour qu'un autre algorithme puisse être implanté en utilisant la même architecture, il suffira de générer un autre code. L'architecture une fois spécifiée, toute une classe d'applications pourra s'adapter à cette architecture. Pour une application donnée, il suffi, donc de ne générer que le code correspondant.

Cette nouvelle approche présente l'avantage de générer une architecture à partir d'une description logicielle, et de pouvoir par la suite améliorer cette architecture en suivant un certain nombre de critères de performance (surface, vitesse,...): on aura donc le schéma suivant:

Un premier flot de synthèse permet de générer une première architecture de chemin de données avec le code correspondant pour le contrôle des opérations. Cette même architecture pourra s'adapter à une nouvelle application; il suffira d'avoir recours à un outil de génération de code permettant d'avoir le code correspondant. Par ailleurs la tâche à effectuer s'apparente à une allocation sous contrainte de ressources limitées.

L'objectif visé est donc d'étendre la synthèse architecturale à la génération d'architectures reprogrammables. Afin de bien mener cette tache, il faut:

- définir un modèle d'architecture de la partie contrôle pour une génération automatique,
- adapter le flot de synthèse pour générer la nouvelle architecture: utiliser la sortie de la synthèse pour générer un code chargeable dans cette architecture.

La conception d'un circuit VLSI dans un environnement de synthèse de haut niveau part d'une description comportementale. Cette description sera compilée par un synthétiseur de haut niveau, dont le rôle consiste à générer une description RTL (*Register Transfer Level* : Niveau transfert de registres) réalisant d'une part le comportement donné, et satisfaisant d'autre part les contraintes de spécification. Le circuit au niveau transfert de registres est

généralement composé principalement d'un contrôleur et d'une partie opérative. Plusieurs architectures sont possibles pour le contrôleur. On se penchera ici sur une architecture reprogrammable pour celui-ci.

Dans la suite de ce papier, on présentera dans le chapitre suivant l'architecture globale du contrôleur. Dans le chapitre 3, la méthode utilisée sera illustrée par un exemple. Les résultats des comparaisons seront donnés dans le chapitre 5, et on terminera par une conclusion dans le chapitre 6.

2. Architecture cible pour le contrôleur

La partie opérative fournie au niveau RT

conditions calculées dans la partie opérative

4: d'un registre adresse "ADR" stockant temporairement l'adresse calculée de la nouvelle commande,

La partie opérative contient un registre condition "RCC" mémorisant toutes les conditions calculées dans la partie opérative. Le bit significatif de ce registre va indiquer la position de la prochaine commande dans la ROM.

est obligé de temporiser pour attendre les comptes-rendus de la partie opérative.

Une étape de contrôle s'exécute sur trois périodes d'horloge (figure 3). A chaque front montant, en fonction de l'adresse de la transition courante (ADR), la partie ROM du contrôleur génère le vecteur de commandes (CD). Pendant ce temps, le séquenceur, en fonction de la valeur donnée par la logique des codes condition (CC) et calculée le cycle précédent, et en fonction de la nature de l'état suivant, fournit l'adresse de la transition suivante qui ne sera exploitée que le cycle suivant. En parallèle, la partie opérative, exécute les opérations correspondant au vecteur de commandes (CD) généré par la partie contrôle au cycle précédent.

Dans le cas d'une opération conditionnelle, la nouvelle transition au niveau de la ROM, au cycle i, dépend des résultats des opérations exécutées dans la partie opérative pendant le cycle i-2 (pipe=2).

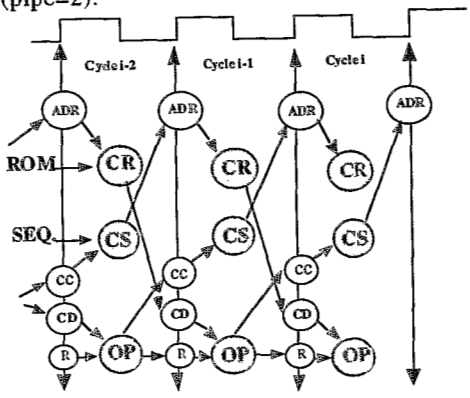

Figure 3: Synchronisation des opérations entre parties opérative et contrôle

CC: Code Condition, **R:** Valeur de registres,
CD: Commandes, **CR:** Contrôle: partie ROM,
CS: Contrôle: partie Séquenceur,
ADR: Adresse d'une transition dans la

1- Les commandes envoyées par la partie contrôle vers la partie opérative et les comptes-rendus (CC) retournés par celle-ci doivent être mémorisés (pendant un cycle d'horloge) afin de bien synchroniser les événements entre les deux parties: c'est le rôle des registres code condition (RCC) et micro-instruction (voir figure 2).
2- En cas de dépendance entre instruction et condition, l'insertion d'états intermédiaires est alors nécessaire. Ces états sont des états de repos (pour le contrôleur) permettant d'attendre les comptes-rendus adéquats de la partie opérative pour le calcul de l'adresse de la prochaine transition. L'état intermédiaire est inséré automatiquement lors de l'ordonnancement ("Scheduling").

L'exécution d'une opération passe donc par trois cycles fondamentaux:

CYCLE 1	CYCLE 2	CYCLE 3
RECHERCHER	**DECODER**	**EXECUTER**
SEQUENCEUR	ROM	P.OP.

Figure 4: Cycles d'exécution d'une opération

-Cycle 1: Calcul de l'adresse de la prochaine instruction,
-Cycle 2: Décodage de l'instruction: lecture de la ROM,
-Cycle 3: Exécution de l'instruction par les éléments de la partie opérative.

L'attente du cycle 1 est nécessaire si l'adresse de la nouvelle opération dépend des résultats de l'opération précédente (opération conditionnelle: voir figure 5). Dans le cas contraire, Le calcul de l'adresse de la nouvelle opération ne demande pas d'attente, car cette adresse est indépendante du calcul qui se fait dans la partie opérative. Si la prochaine opération n'est pas conditionnelle, deux cycles seulement sont nécessaires, un cycle de recherche de l'opération, et un cycle de décodage (lecture de la ROM). L'attente

3. Exemple de Conception

Pour illustrer cela nous allons prendre un exemple. Soit la description en VHDL d'un circuit qui calcule le GCD (Greatest Common Divider : PGCD) de 3 nombres entiers positifs x, y, z:

```
ENTITY GCD3 IS
 PORT (start: IN BIT;
       xi, yi, zi: IN INTEGER;
       ou      : OUT INTEGER);
END GCD3;
ARCHITECTURE behavior OF GCD3 IS
 BEGIN PROCESS
   VARIABLE x, y, z: INTEGER;
   BEGIN
    WAIT UNTIL (start='1');
    x:=xi; y:=yi; z:=zi;                1
    WHILE (x/=y) LOOP
      IF (x<y) THEN y:=y-x;             2
               ELSE x:=x-y;             3
    END IF; END LOOP;
    WHILE (y/=z) LOOP
      IF (y<z) THEN z:=z-y;             4
               ELSE y:=y-z;             5
    END IF; END LOOP;
    ou<=y;                              6
 END PROCESS;
END behavior;
```

Figure 5: Description VHDL de l'algorithme du "GCD3"

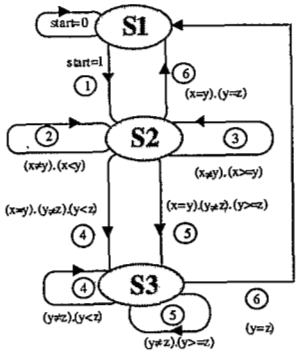

Figure 6: Automate de contrôle

La description initiale sera modifiée après l'ordonnancement (*Dynamic Loop Scheduling* : Ordonnancement à boucles dynamiques [2]) de façon à regrouper toutes les conditions dans un seul registre (registre code condition : RCC). A chaque test une condition est calculée sous forme de variable booléenne de un bit. Tous ces calculs se feront donc dans la partie opérative.

Dans cet automate (voir figure 7), des transitions de repos ont été introduites pour respecter les contraintes vues dans la section 2: les transitions T2, T6 et T11 ont été introduites pour attendre que les opérations sur C0, C1 et C2 s'exécutant dans la partie opérative soient terminées. Les transitions T3.0, T3.1, T7.0, T7.1, T7.2, T12.0, T12.1 et T12.2 permettent d'attendre le calcul, par le séquenceur, des adresses suivantes.

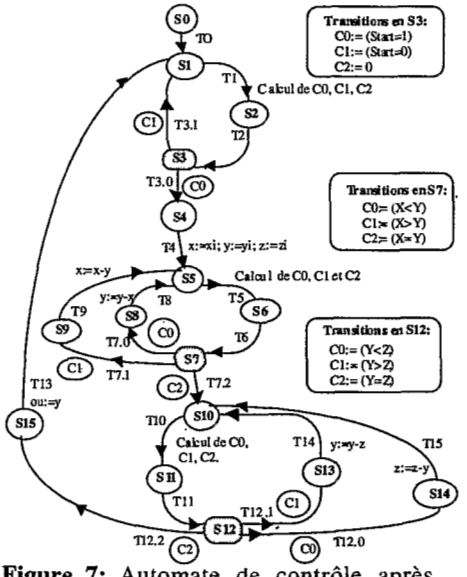

Figure 7: Automate de contrôle après introduction des cycles de calcul des conditions

Dans cet exemple, on a trois conditions à tester (C0, C1 et C2) à chaque fois: donc le registre RCC sera de trois bits et la sortie de la logique des conditions sera de deux bits. En général, il faut une logique (encodeur) à p entrées et n sorties: n=[log(p)/log(2)]+1.

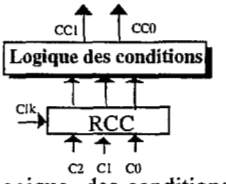

Figure 8: logique des conditions

Dans notre cas la sortie de cette logique donnera la position du bit Ci: si C0=1, la sortie sera de 0 (CC=0), si C1=1, la sortie sera de 1 (CC=1), si C2=1, la sortie sera de 2 (CC=2).

Dans cet automate, on remarque qu'il y a deux types d'états: Les états simples d'où sortent une seule transition: S0, S1, S2, S4, S5, S6, S8, S9, S10, S11, S13, S14, S15, et les états complexes d'où partent des transitions multiples: S3, S7, S12.

Les états simples correspondent à des branchements inconditionnels (ou à un séquencement ordinaire) et les états complexes correspondent à des branchements conditionnels.

On aura donc deux types de transition (correspondant à deux modes d'adressage):

1- branchement conditionnel: (transition multiple)
==> Mode= 1
2- branchement inconditionnel: (transition simple)
==> Mode= 0

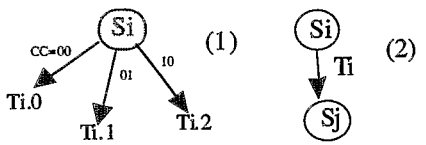

Figure 9: Différents types de transitions (multiples (1), simples (2))

Transition	Operations	Transiotn svt.	CC	Mode
T0	NOP	T1	XX	0
T1	C0, C1, C2	T2	XX	0
T2	NOP	T3	XX	1
T3.0	NOP	T4	00	0
T3.1	NOP	T1	01	0
T4	X, Y, Z	T5	XX	0
T5	C0, C1, C2	T6	XX	0
T6	NOP	T7	XX	1
T7.0	NOP	T8	00	0
T7.1	NOP	T9	01	0
T7.2	NOP	T10	10	0
T8	Y:= Y-X	T5	XX	0
T9	X:= X-Y	T5	XX	0
T10	C0, C1, C2	T11	XX	0
T11	NOP	T12	XX	1
T12.0	NOP	T15	00	0
T12.1	NOP	T14	01	0
T12.2	NOP	T13	10	0
T13	ou:= Y	T1	XX	0
T14	Y:= Y-Z	T10	XX	0
T15	Z:= Z-Y	T10	XX	0

Figure 10: Table de transitions

Pour notre exemple, la table de transitions est résumée dans la figure 10. Les transitions de repos seront représentées dans cette figure par des actions nulles (NOP: pas d'opérations), dans la figure suivante (10), elles seront représentées par une suite de 0. Les transitions T1, T5 et T10 concernent le calcul à chaque test des codes de condition.

Adresses		Operations	Transition svt.	Mode
(0) 00000	T0	00000000000	T1 00001	0
(1) 00001	T1	C0, C1, C2	T2 00010	0
(2) 00010	T2	00000000000	T3 00011	1
(3) 00011	T3.0	00000000000	T4 00101	0
(4) 00100	T3.1	00000000000	T1 00001	0
(5) 00101	T4	X, Y, Z	T5 00110	0
(6) 00110	T5	C0, C1, C2	T6 00111	0
(7) 00111	T6	00000000000	T7 01000	1
(8) 01000	T7.0	00000000000	T8 01011	0
(9) 01001	T7.1	00000000000	T9 01100	0
(A) 01010	T7.2	00000000000	T10 01101	0
(B) 01011	T8	Y:= Y-X	T5 00110	0
(C) 01100	T9	X:= X-Y	T5 00110	0
(D) 01101	T10	C0, C1, C2	T11 01110	0
(E) 01110	T11	0000000000	T12 01111	1
(F) 01111	T12.0	0000000000	T15 10100	0
(10) 10000	T12.1	0000000000	T14 10011	0
(11) 10001	T12.2	0000000000	T13 10010	0
(12) 10010	T13	ou:= Y	T1 00001	0
(13) 10011	T14	Y:= Y-Z	T10 01101	0
(14) 10100	T15	Z:= Z-Y	T10 01101	0

Figure 11: Contenu de la ROM

4. Structure du séquenceur et de la ROM

Le séquenceur calcule l'adresse suivante en fonction de la nature de chaque transition. Si la transition concernée est simple (mode=0), l'adresse suivante sera soit la prochaine ligne de la ROM soit une adresse quelconque dans le cas d'un branchement inconditionnel. S'il s'agit de transitions multiples (mode=1), l'adresse de la transition suivante sera calculée en ajoutant au champ "adr svt" les codes condition ("adr. svt."+CC).

Toutes les transitions issues d'un même état seront rangées séquentiellement en fonction du code condition CC. Le type d'adressage (ou mode) sera fonction de la nature de la transition suivante (simple ou multiple) (figure 10).

Chaque transition est implantée dans une ligne de la ROM (figure 11). Une première partie de la ligne (représentée à gauche) donne les signaux de sortie pour la partie opérative, une deuxième partie (à sa droite) donne l'adresse de la micro-

instruction suivante et définit la nature de cette micro-instruction (transition simple ou multiple).

5. Résultats

La taille de cette partie contrôle a été fixée par la sortie du synthétiseur utilisé dans cette exemple (AMICAL: synthétiseur architectural développé à TIMA/INPG de Grenoble (FRANCE)). La ROM contient 21 mots de 59 bits.

Le gain en temps se verrait avec des exemples beaucoup plus grands. Le gain est plutôt beaucoup plus dans la flexibilité et le temps de conception du circuit.

Une synthèse RTL a été aussi réalisée avec SYNOPSYS [10]. Ceci nous a permit de calculer le nombre total de cycles nécessaires à l'exécution de l'exemple pour les nobres: (10, 25, 20) et ceci dans les deux architectures possibles

		câblée	programmable	Différence en %
Surface	Contrôleur	0.125 mm2	0.95 mm2	+660 %
	P. opérative	2.53 mm2	6.13 mm2	+149.29 %
	Total	2.655 mm2	7.08 mm2	166.66 %
Temps de cycle		25 ns	70 ns	+180 %
Nombre de cycles		16	39	+143.75 %

Table 1: Estimation de la surface et du temps pour les deux types d'architecture

La taille du champ "Action" du registre micro-instruction est de 53 bits (fixée par la taille de la partie opérative), celle du champ "Adr. svt." est de 5 bits fixée par le nombre total de transitions (21) de l'automate de contrôle (voir figure 7).

L'architecture utilisée est à base de bus (l'architecture câblée avec un "pipe" de 1). Les résultats sont donnés dans la table 1 (dernière ligne). L'exemple ("GCD3") a été aussi exécuté (en C) sur machine SPARC 20 (horloge de 50 MHz) pour faire une comparaison entre la solution logicielle et celle matérielle, cette comparaison est résumée dans la table 2.

En moyenne, pour notre exemple, la solution matérielle programmable est

Une implémentation reprogrammable est, en générale, 50 à 100% [6] plus large que celle câblée. Par contre le temps est mieux contrôlé dans une architecture microprogrammée: ceci est du au fait qu'il ne dépend, en grande partie, que du temps d'accès à la ROM. Si on veut implémenter un nouveau comportement, le temps ne change que légèrement, par contre avec une architecture sous forme de FSM, ce temps change considérablement [5].

L'avantage d'une architecture reprogrammable réside dans la régularité et la flexibilité. Le choix entre ce type d'architectures et une architecture sous forme de FSM (câblée) dépend principalement de la taille de l'application: pour de petites applications, l'architecture câblée est souvent conseillée, parce que simple, rapide et occupe moins d'espace. Pour les applications de grande taille et complexes, une architecture microprogrammée serait plus intéressante [5].

L'architecture étant générée, il s'agit maintenant de voir comment l'adapter sur d'autres exemples (d'autres applications du même domaine). Des calculs de performances doivent être faits pour déterminer si cette architecture répond à certains critères (vitesse, surface,...). On pourrait être amener à réduire le nombre de registres, par exemple, vue que la surface obtenue est trop importante, dans ce cas l'ordonnancement doit être refait en tenant compte de ces contraintes (figure 1).

Le but est de fixer une architecture optimale qui pourrait par la suite être utilisée pour une large gamme d'applications du même domaine. Pour une seconde application, il suffit juste de générer un code à partir de l'algorithme décrivant cette application en utilisant les techniques de génération de code.

Remerciements

Nous tenons à remercier vivement tous les chercheurs du laboratoire TIMA/INPG de Grenoble pour nous avoir permis de mener à bien ces travaux et principalement: Maher Rahmouni, Clifford Liem, P. Vijay Raghavan, Hong Ding, Mohamed Romdhani et Elisabeth Berrebi.

Références:

[1] P.G. Paulin, C. Liem, T. C. May et S. Suturwala, "Code Syn: A Retargetable Code Synthesis SysteM", 7th International High-Level Synthesis Wokshop, 94.

[2] M. Rahmouni, K. O'Brien et A.A. Jerraya, "A loop-based Scheduling Algorithm For Hardware Description Languages", Parallel Processing Letters, Vol. 4 No 3 (1994), pp 351-364.

[3] Jessi AC8 project, Subproject 6: "RTL synthesis, Workpakage 6.1: Control dominated RTL synthesis ", CSI/INPG, Grenoble, Mai 94.

[4] M. Aichouchi, "Etude des liens entre la synthèse architecturale et la synthèse au niveau transfert de registres", thèse de Docteur, TIMA/INPG, Juin 94.

[5] M. Froidevaux et J.M. Gentit, "MPEG1 and MPEG2 system layer Implementation Trade-off between micro coded and FSM architecture", Thomson Consumer Electronics Components , Meylan France, ICCE, Chicago, Juin 95.

[6] J. Bhasker et H. C. Lee, "An optimizer for Hardware Synthesis", IEEE Design and Test of Computers, pp. 20-36, 90.

[7] P. Paulin, J. Fréhel, E. Berrebi, C. Liem, J. C. Herluison et M. Harrand, "High-Level Synthesis and Codesign Methods: An Application to Videophone Codec", Euro-VHDL, Brighton, Sep. 95.

[8] G. De Micheli, "Computer-Aided Hardware-Software Codesign", IEEE Micro, Aout 94.

[9] G. Goossens, J.V. Praet, D. Lanneer et W. Guerts, "Programmable Chips in Consumer Electronics and Telecommunications", NATO Advanced Study Institute on Hardware/Software Co-Design, Tremezzo, Italy, Juin 95.

[10] "Synopsys VHDL System Simulator (VSS) Tutorial", Version 3. 0b, Juin 93.

[11] M. Benmohammed , A. A. Jerraya, P. Kission et M. Rahmouni, «Génération d'Architectures microprogrammées dans un Environnement de Synthèse de Haut niveau», revue internationale des technologies avancées, Algerie, No: 8, Dec. 95.