INSA TOULOUSE

Projet Systèmes Informatiques

Du compilateur vers le microprocesseur

Yann LABEL
Rémi SAUREL
4 IR TD B2

Enseignants :
Daniela DRAGOMIRESCU
Eric ALATA
Benoit MORGAN
Carla SAUVANAUD

Sommaire

Partie 1 : le compilateur

Les premières séances de TP auront servi à la prise en main de Lex et Yacc, avec l'écriture de règles lexicales et syntaxiques de base. Notre objectif était initialement d'avoir rapidement une analyse sommaire du code source en entrée, pour pouvoir ensuite étoffer à partir de ce premier jet. Les règles de bases pour un programme minimal permettaient d'analyser un programme de la forme :

```
void main() {
      int var1, var2;
      var2 = 3 * 2 + var1;
}
```

1) La table des symboles

La deuxième étape a été de fournir une ébauche de table des symboles. Ici, le choix a été fait que les conteneurs utilisés (listes, piles) soient statiques, c'est à dire sans allocation dynamique. Pour cela, des constantes (puissance de 2 pour la beauté du geste) définissent la capacité des conteneurs (par exemple, 8192 entrées disponibles dans la table des symboles).

Rétrospectivement, ce choix semble avoir été le bon, puisque la consommation en mémoire du compilateur tourne en dessous d'une vingtaine de Mo sur un système 64 bits, ce qui est largement acceptable de nos jours. De plus, cela nous a permis de nous consacrer d'avantage à la logique du compilateur, tout en assurant de très bonnes performances (les seules allocations mémoires effectuées par le programme sont destinées à copier les chaînes de caractères fournies par Lex, et pour la sortie assembleur). Les limites fixées l'ont été en jetant un œil au document de référence standardisant le langage C (un brouillon est accessible ici, section 5.2.4.1 "Translation limits"), ce qui nous permet *a priori* d'analyser n'importe quel programme raisonnablement dimensionné.

Un symbole de la table est défini comme une struct C contenant un nom et une adresse. Par la suite, nous avons ajouté un booléen mémorisant si la variable a été initialisée avant sa première utilisation et un champ type.

Le type d'une variable est constitué d'un type de base (void, char, int) et de jusqu'à 63 niveaux d'indirections (1 indirection équivaut à un pointeur, 2 indirections à un pointeur de pointeur etc.). Pour chaque niveau d'indirection, il est possible de spécifier ou non le mot clé `const` :

```
int a1; const int a2; int *a3; int * const *a4;
```

L'implémentation du langage C nous a démontré la pertinence des limites évoquées plus haut : étant un langage au départ taillé pour la vitesse, mais aussi `pour` la vitesse de compilation, il était nécessaire que la logique du compilateur reste simple et efficace. Pour certaines de ces limites, les qualificateurs const notamment, pouvoir agir directement sur les bits d'un seul entier pour stocker 64 informations différentes est une bonne optimisation temporelle et spatiale du compilateur.

Un symbole déclaré par le programmeur aura forcément un nom, tandis qu'un symbole temporaire (utilisé pour calculer une expression arithmétique, par exemple) aura un nom particulier. Pour déterminer ce nom, nous avons là encore regardé le standard C qui spécifie dans 7.1.3 "Reserved identifiers" que tous les noms d'identificateurs d'un programme commençant par un underscore "_" sont réservés pour l'implémentation (en l'occurrence, nous), et que le non respect de cette règle et les conséquences qui en résultent sont uniquement à la charge du programmeur ("c'est pas notre problème").

2) Les conditions et les boucles

Nous nous sommes ensuite concentrés sur l'implémentation des conditions (if-else). Nous nous sommes rapidement aperçus qu'un système de "labels" était nécessaire, puisque les sauts requis pour ne pas exécuter certaines instructions sont transcrits en assembleur parfois sans que leur destination ne soit connue.

Cela implique une deuxième passe de compilation, qui va réécrire certaines parties du code généré après que toutes les destinations de saut soit connues. Le mode opératoire a été d'écrire en permanence le code en mémoire, avec des "placeholders" pour les adresses inconnues. La chaîne choisie pour garder la place est "000000", elle autorise un programme de dix millions d'instructions. Une fois tout le code généré, il suffit de rechercher-remplacer cette chaîne par les bonnes adresses correctement mémorisées. Pour faciliter la génération de code, nous avons écrit une fonction assemblyOutput qui fonctionne sur le même principe que les dérivés de printf pour formatter le texte.

Le tristement célèbre conflit shift/reduce provoqué par nos règles Yacc avec le if-else nous a fait prendre conscience de la nécessité de surveiller la machine à états générée. En ajoutant le flag -v lors du passage à travers Yacc, nous avons pu comprendre beaucoup de choses sur le fonctionnement du parseur et nous avons pu résoudre tous les conflits, même ceux qui sont apparus plus tard après adjonction de règles supplémentaires (la sortie fait désormais plus de 8 000 lignes, difficile de s'y retrouver !).

L'ajout des boucles while, do-while et for était la suite logique après cette étape. Nous avons dans un premier temps tenté de nous baser sur le système de labels des conditions. Utiliser la même pile c'est rapidement révélé impossible, surtout une fois les fonctionnalités de break et continue implémentée : un if imbriqué dans une boucle était impossible par exemple. La solution a été de créer une deuxième pile de labels dévolue aux boucles. Le fonctionnement des 3 boucles est très similaire, avec plus de complexité pour la boucle for et ses phases initialisation/test/action/post-action.

3) Les fonctions

Les fonctions étaient la grande étape suivante. Pour pouvoir les implémenter correctement, nous avions besoin d'un pointeur de pile et d'un pointeur mémorisant l'adresse de retour une fois l'appel terminé.

Dès lors, nous ne pouvions plus nous satisfaire du jeu d'instruction fourni et nous avons ajouté deux instructions :

- `STK` ***`op mode`*** : permet de manipuler le stack pointer, en y ajoutant la valeur ***`op`*** quand ***`mode`*** vaut 0 (un autre mode est utilisé pour les pointeurs, il sera détaillé plus tard). On décrémente le pointeur simplement en spécifiant un ***`op`*** négatif.
- `RET` : retourne à l'adresse stockée en première position dans le cadre de pile courant.

D'abord sans paramètres ni type de retour, les fonctions se sont vues améliorées jusqu'à reproduire entièrement le comportement du langage C sur ce point (exception faite des fonctions à nombre variable d'arguments). Les vérifications du nombre et du type des arguments est faire à la compilation, avec une erreur détaillée dès qu'une incohérence se présente.

Le système de prototype de fonctions a été incorporé, ce qui permet d'utiliser une fonction seulement déclarée pourvu qu'elle soit définie quelque part dans l'unité de compilation. Là encore, les vérifications sont faites pour que le prototype concorde avec la définitions et que s'il y a plusieurs prototypes, qu'ils soient identiques.

Un gros effort a en effet été effectué pour que notre compilateur propose des diagnostics aussi précis et complets que possible, partout où cela est nécessaire. Cela vaut autant pour une variable non déclarée, qu'un paramètre incorrect ou la modification d'une variable constante, l'affectation d'un pointeur avec un nombre, l'inverse, etc.

La convention d'appel choisie pour les fonctions est assez classique :

- La première adresse de la pile est l'adresse de retour vers l'appelant ;
- La deuxième est réservée à la valeur de retour s'il y en a une (fonction autre que void) : si tel est le cas, à charge de l'appelant d'aller copier cette valeur (initialisée par les return) pour l'exploiter ensuite ;

- Les arguments sont ensuite empilés, s'il y en a. Ils sont récupérés en tant que variables locales du même nom que les paramètres de la fonction ;
- Les variables locales de la fonction viennent ensuite.

Pour ce qui est de l'accès aux variables locales, leur adresse est corrigée à l'exécution par le stack pointer, pour bien respecter le principe de la pile d'appel.

4) Les pointeurs

Un point assez ennuyeux à gérer a été la combinaison des appels de fonction avec les pointeurs. En effet, initialement nous avons utilisé l'adresse dans la table des symboles comme adresse de l'objet à l'exécution ; cela ne pouvait évidemment pas marcher quand une fonction était appelée avec pour paramètre un pointeur venant du cadre de pile précédent : on modifiait en fait une variable locale au lieu de la bonne.

C'est ici que le deuxième mode de l'instruction STK se fait utile : en appelant "STK op 1", on ajoute à op la valeur du stack pointer, ce qui rend son contenu absolu et non plus relatif s'il s'agissait d'une adresse.

Deux instructions supplémentaires ont fait leur apparition ;

- `DR1 op1 op2`, équivalente au pseudo-code suivant :

```
memory[memory[op1 + stackPointer]] = memory[op2 + stackPointer]
```

- DR2 op1 op2 , équivalente au pseudo-code suivant :

```
memory[op1 + stackPointer] = memory[memory[op2 + stackPointer]]
```

Elles permettent de considérer une adresse mémoire comme contenant une autre adresse (un pointeur, autrement dit) et non plus une valeur. On voit que la correction "`+ stackPointer`" n'est pas faite pour le contenu du pointeur, puisqu'on l'a rendu absolu avec `STK op 1` précédemment.

Les variables globales sont également gérées par le compilateur, et leur accès se fait également par adresses absolues.

Nous avons choisi d'implémenter les blocs d'imbrication de code : nous pouvons, comme dans le if et le while par exemple, entourer plusieurs instructions d'accolades pour ouvrir une nouvelle portée. Les variables qui y sont déclarées peuvent avoir le même nom que les variables du bloc englobant (ou que les variables globales), la résolution d'une variable se fait donc en cherchant d'abord dans le bloc le plus imbriqué, et en remontant d'un jusqu'aux globales. Une variable qui n'a été trouvée nulle part est considérée comme non déclarée.

5) Les tableaux

Une fois les pointeurs correctement implémentés, nous avons décidé de continuer avec les tableaux. Un tableau est défini comme une suite de cases contiguës, ainsi que par un pointeur (constant) vers la première case, qui sera manipulée par l'utilisateur. Les cases du tableau sont, à l'image des symboles temporaires, des symboles particuliers avec un nom réservé qui permet au compilateur d'accéder rapidement à une case particulière. L'initialisation d'un tableau est gérée à la déclaration, avec la syntaxe `int tab[N] = {1, 2, …};` On peut spécifier ou non la dimension, dans ce cas c'est le nombre d'éléments dans la liste d'initialisation qui fait foi pour la taille. Si on précise la taille, on peut spécifier autant d'éléments que de cases, moins (ceux non précisés sont initialisés à 0) ou plus (dans ce cas, erreur de compilation).

Basées sur le tableau, les chaînes de caractères les ont suivi de près. Nous avons voulu implémenter la méthode de "string interning" qui consiste à ne conserver qu'une seule copie de toutes les chaînes au contenu identique. Tolérée mais non requise par le standard C, cette méthode nous a paru être intéressante à tester. L'implémentation réalisée est simple : on crée le tableau en mettant comme

nom de symbole le contenu de la chaîne. Si on trouve un symbole avec le même nom, on se contente de le réutiliser.

La suite logique a été de modifier l'instruction PRI pour qu'elle puisse afficher du texte. Nous avons simplement rajouté un paramètre, à l'image de STK. L'opérande supplémentaire pilote le comportement à l'exécution, et une valeur de 0 affichera un entier, 1 affichera un pointeur (adresse entière affichée en hexadécimale, similaire au %p de printf), 2 affichera un simple caractère (interprétation de la valeur comme un point de code ASCII), et 3 affichera une chaîne de caractère (terminée par l'octet NUL comme en C).

6) Autres fonctions

D'autres éléments ont ensuite été rajoutés au compilateur :

- Conditions ternaires (a ? b : c) ;
- Opérateur sizeof, qui sert principalement à déterminer le nombre de cases d'un tableau puisque les int, char et pointeurs ont tous une taille de 1 ;
- Valeurs énumérées (`enum MonEnum {ValeurEnum1, ValeurEnum2};`). On ne peut déclarer des variables de type énuméré qu'en utilisant la syntaxe `enum NomEnum = ValeurEum11;`, car le système de typedef n'a pas été implémenté ;
- Les blocs switch. Ils ont été plutôt compliqués à intégrer, il fallait gérer le cas default, le break facultatif etc., mais il fonctionne bien selon nos tests ;
- Les instructions `goto` qui permettent de piloter le flot de contrôle à loisir ;
- La lecture d'une valeur entière depuis stdin avec l'instruction SCN op qui demande une valeur et l'écrit dans op après que l'utilisateur ait appuyé sur Entrée. Cela autorise des programmes interactifs.

L'ajout de 5 opcodes s'est fait en supprimant l'opération SUP (redondante avec INF), nous voulions en effet ne pas interdire leur codage sur 4 bits. Les instructions sont au nombre de 16 numérotées de 0x0 à 0xF.

Les opérations arithmétiques +, -, *, /, ++, *= etc. ont été correctement gérées, et sont cohérentes avec la nature des variables (on peut ajouter un scalaire à un pointeur mais on ne peut pas ajouter deux pointeurs entre eux par exemple).

Dernier élément qui a complètement changé depuis l'apparition des pointeurs : la manière dont sont gérés les variables. Puisque une variable peut être déréférencée, comme dans int a = *ptr; ou dans (*ptr)++, une nouvelle structure a fait son apparition, dereferencedSymbol_t, qui combine un symbole avec une ou plusieurs "étoiles". Cela permet par exemple aux affectations de se faire à la bonne adresse ou avec la bonne valeur à l'aide des opcodes DR1 et DR2.

7) L'interpréteur

L'interpréteur demandé pour le projet a également été réalisé, ce qui s'est avéré indispensable pour tester et corriger la génération de code. Nous avions mal lu l'exigence de créer cet interpréteur avec Lex et Yacc, c'est pourquoi il se résume à un fichier C de 250 lignes qui lit les opcodes et opérandes avec les fonctions classiques d'entrée standard.

Dans le fichier assembleur en sortie, toujours dans une logique de débug, nous avons placé des commentaires sur le code généré. Cela nous a permis de nous repérer rapidement dans l'assembleur. Notre interpréteur les ignore (ils sont en fin de ligne après un point-virgule), mais un simple `#define STRIP_COMMENTS` dans le fichier source "assembly.h" permet de ne pas les générer, pour un fichier plus léger ou un interpréteur moins tolérant que le nôtre. On peut également écrire les opcodes sous forme de texte (ex : `COP 0 1` au lieu de `5 0 1`) pour se repérer, avec le `#define OPCODE_TEXT` dans le même fichier. En revanche, le code généré par le compilateur sera dans ce cas mal exécuté, notre interpréteur attendant toujours les nombres.

Partie 2 : le microprocesseur

La seconde partie du projet consistait en la réalisation d'un microprocesseur de type RISC avec pipeline.

La cible était un carte FPGA de type Spartan, et nous avons utilisé le langage de description VHDL.

Il est intéressant de noter que le processeur que nous allons vous présenter maintenant n'est pas compatible avec le compilateur que nous avons vu en première partie. En effet, notre microprocesseur est orienté registre, tandis que le compilateur est orienté mémoire, engendrant un fonctionnement complètement différent malgré des instructions similaires.

1) Notre démarche

Comme conseillé, nous avons adopté une démarche par étape. Dans un premier temps nous avons réalisé l'ensemble des composants. Après les avoir testé individuellement, nous avons commencé l'intégration, en ajoutant les composants requis selon les instructions que nous implémentions. En complexifiant notre chemin de données, nous sommes arrivés peu à peu à l'objectif.

Il faut cependant noter que nous n'avons pas eu le temps d'implémenter et de tester les sauts (asm : JMF JMP).

2) Les choix d'implémentation

Autant que possible, nous avons privilégié une approche asynchrone pour les composants. En effet, les différents pipeline du chemin de données sont synchrone. La temporisation que cela implique nous a dispensé de rendre l'ensemble des composants synchrone.

En définitive, seul le compteur IP, la gestion des aléas et les écritures en mémoire sont synchrones.

D'autre part, au risque de perdre en généricité, nous avons fait le choix de créer un module pour chaque composants. Il était possible d'instancier plusieurs fois le même module et d'adapter son comportement selon le câblage. Notre choix nous a permis d'avoir plus de clarté dans nos phase de test et de débug.

Ce choix à été particulièrement intéressant lors de l'implémentation de la gestion des aléas. En effet le "NOP" injecté dans le chemin de donné à chaque détection d'aléas est en fait traité dans le cas par défaut. Ci-après un exemple VHDL :

```
with sel select
      S <=   IN_1 when AFC, --On propage la sortie 1 pour toutes les op d'écriture
                    IN_1 when STORE,
                    IN_2 when others; --On propage la sortie 2 dans tout les autres
cas, y compris pour les NOP
```

En procédant ainsi, il n'était plus nécessaire de définir l'ensemble des opérations par des constantes, le code devient dès lors plus léger et requiert moins de ressources :

```
constant AFC : STD_LOGIC_VECTOR(7 downto 0) := "00000110" ;
constant STORE : STD_LOGIC_VECTOR(7 downto 0) := "00001000" ;
      --constant LOAD : STD_LOGIC_VECTOR(7 downto 0) := "00000111" ; inutile de le
définir
```

La gestion des flags de l'ALU nous a posé quelques problèmes. En effet, les calculs dans l'ALU se font de manière asynchrone. Nous avions au début implémenté un process qui scrutait les modification du buffer stockant le résultat des opérations.

Cependant, cette solution n'était pas synthétisable. Nous avons donc opté pour une solution asynchrone pour la gestion des flags :

```
Negatif <= N when buff(8) = '1' else no_flag ;
        Overflow <= O when (op1(7)=op2(7) and op1(7)/=buff(7)) else no_flag;
        Zero <= Z when buff= MOT_ZERO else no_flag ;
        Carry <= C when (buff(8)='1') and (ctr_ALU = ADD or ctr_ALU = SUB) else
no_flag ;
        flag <= std_logic_vector(Negatif + Overflow + Zero + Carry) ;
```

Il existe donc des moments (brefs mais réels) ou la valeur des flags n'est pas exacte. Cependant, ces instants sont si court qu'ils ne gênent pas lors de l'exécution.

3) Problèmes

Nous avons été confronté à un problème lors de la gestion des aléas.

En effet, nous avons dans un premier temps réalisé un composant "à cheval" sur les deux premiers étages du chemin de données. L'idée était de détecter (dans un cas précis, à savoir l'instruction "AFC") si les opérandes de deux instructions consécutives entraient en conflit.

Cette méthode s'est révélé très limitée. Elle ne nous permettait pas de détecter les aléas de deux instructions non consécutives.

Nous avons alors mis en place une petite logique de contrôle.

Dans un premier temps, nous avons réalisé qu'un aléa ne se présente que lorsqu'on utilise les instructions AFC, LOAD et COP.

Nous avons aussi noté que si une instruction utilise un registre en cours de modification, le résultat constitue un aléas, et ce même si les instructions ne sont pas consécutives. Pour être plus précis, le code suivant génère un aléas qu'il faut être capable de détecter :

```
AFC R1 10
ADD R2 R3 R3
ADD R2 R1 R1 --ici il y aura un aléa dû au registre R1 encore en cours de modifica-
tion.
```

Pour ce faire nous utilisons un tableau indexé par les numéro de registres :

```
type BANC_TYPE is array (15 downto 0) of STD_LOGIC_VECTOR (1 downto 0) ;
signal alea_tab : BANC_TYPE ;
```

Ce tableau contient en fait le nombre de bulles à ajouter dans le chemin de données, en fonction du registre qu'on scrute.

À chaque fois qu'on "lis" une instructions sensible ("COP AFC LOAD"), on ajoute la valeur 3 dans ce tableau, à l'indice du registre cible. Dans l'exemple ci-dessus, on aurait : aleas_tab[1]<="11" lors de l'execution de la première instruction.

À chaque cloque d'horloge, on vérifie la valeur dans le tableau pour chacun des registres utilisé dans l'instruction (en position 2 et 3). Ex:

```
ADD R2 R3 R4 -- Cette instruction génère une lecture de aleas_tab[3] et
aleas_tab[4].
```

Si les valeurs lues ne sont pas nulles, alors on génère des NOP.

Bien entendu, l'ensemble des valeurs non nulles est décrémenté à chaque tic d'horloge.

Afin d'arrêter l'exécution en amont, nous avons choisit de générer un deuxième signal d' horloge pour le compteur IP, et le premier pipeline. Ce signal est généré dans le module de gestion des aléas.

```
clock <= CK when Inhib = '0' else '0' when Inhib='1';
```

Nous avons retenu cette option afin de ne pas avoir à modifier les autres modules, qui restent donc plus générique et étrangers à la gestion des aléas.

4) La phase de test

Afin de réaliser nos tests, nos avons du rentrer le code binaire des instructions assembleur à la main dans la mémoire d'instructions :

```
signal instruc_zero : std_logic_vector(31 downto 0) := (others => '0') ;
type BANC_TYPE is array (255 downto 0) of STD_LOGIC_VECTOR (31 downto 0) ;
signal banc_instruc : BANC_TYPE :=(
0=> "00000110000000000000100100000000", -- AFC R0 9 -
      1=> "00000101000000010000000000000000", -- COP R1 R0
      3=> "00000001000000100000000000000000", -- ADD R2 R1 R0
      4=> "00000111000000000000000000000000", -- LOAD R0 [0]
      5=> "00001000111111110000000100000000", -- STORE [255] R1
      others => instruc_zero ) ;
```

Les chronogrammes générés nous ont permis de nous assurer du fonctionnement correcte de notre chemin de données. (Vous trouverez avec notre code des fichiers .wcfg, générant des chronogrammes permettant de valider les différentes étapes.)

Une fois le comportement validé, nous avons procédé a la synthèse, en ayant recours à un fichier "user constraint":

```
## Clock signal
NET "clock"         LOC = "V10" | IOSTANDARD = "LVCMOS33";
Net "clock" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100000 kHz;
```

Ce fichier permet de définir les contraintes temporelle du système. Typiquement, notre fréquence de fonctionnement. À la synthèse nous avons donc pu valider que notre fréquence maximale était de 395 MHz, comme le montre le rapport de synthèse ci-après :

```
Timing constraint: Default period analysis for Clock 'CK'
Clock period: 2.531ns (frequency: 395.124MHz)
Total number of paths / destination ports: 80 / 52
```