Compiladors
Especificació de HW–Compilation

Gispert Sánchez, Francesc-Xavier Rodríguez Guasch, Sergio

30 de març de 2015

Índex

1 Introducció

Tradicionalment, aprenem de forma independent i aïllada el funcionament del *hardware* dels computadors i la programació mitjançant *software*. Malgrat les capes d'abstracció que afegim entremig, però, és clar que és tècnicament possible, per a cada programa que puguem expressar mitjançant qualsevol llenguatge de programació, dissenyar un circuit lògic que l'executi.

Aquest projecte té com a objectiu fonamental mostrar que, a més, es pot fer aquest pas d'un programa expressat en un llenguatge de programació convencional a un circuit digital equivalent de forma automàtica.

A tal efecte, dissenyem un llenguatge de programació molt simple de capacitats reduïdes però amb algunes de les estructures de seqüenciació més habituals i construirem un traductor que, a partir dels programes especificats en aquest llenguatge, generi automàticament els circuits corresponents. En particular, aquest llenguatge de programació tractarà amb vectors de bits d'una mida fixada per a cada programa (que, llavors, el programador pot interpretar com a nombres enters, per exemple) i disposarà de sentències condicionals i repetitives, així com d'un sistema de funcions i crides a aquestes. Una de les principals limitacions d'aquest llenguatge, però, serà la falta de memòria (cosa que, d'altra banda, facilitarà la simulació dels programes en circuits).

Tanmateix, l'interès d'aquest projecte no deixa de ser merament acadèmic, ja que els circuits generats no són en general òptims (ni pretenen ser-ho) i, per tant, amb programes d'una complexitat mitjana ja és possible tenir una explosió de portes lògiques que en faci infactible la implementació a la pràctica. Per aquest motiu, un llenguatge limitat com el descrit és suficient per exemplificar la traducció automàtica de programes a circuits.

2 Especificació del llenguatge

El llenguatge que dissenyem serà, en principi, molt simple i limitat. Això ens permetrà concentra-nos en la part més interessant del projecte, que és la de traducció de *software* a *hardware*. De totes maneres, vegem a continuació les principals característiques del llenguatge.

Al principi de cada programa, el programador ha de fixar la mida dels vectors de bits que s'utilitzaran. Llavors, la sintaxi del llenguatge permet aplicar operacions molt senzilles sobre aquests vectors de bits: sumes i restes (interpretant els vectors com a enters codificats en complement a dos); conjuncions, disjuncions i negacions bit a bit; conjuncions, disjuncions i negacions lògiques (entenent que tot vector diferent de zero codifica el valor cert), i comparacions d'igualtat i desigualtat. La precedència dels operadors és la mateixa que a `C`.

Un programa està format per una o més funcions, una de les quals s'ha d'anomenar `main`: aquesta serà, com el seu nom indica, el punt d'entrada a l'execució. Cadascuna d'aquestes funcions pot rebre alguns arguments i ha de retornar exactament un vector de bits com a sortida. A més, les variables utilitzades per una funció han d'haver estat declarades al principi d'aquesta (això facilitarà el procés de traducció, ja que, en traduir el programa a un circuit, cada variable correspondrà a un registre), i una funció només té accés a aquestes variables, així com als seus paràmetres. Una variable especial amb el mateix nom que la funció emmagatzemarà el valor de retorn: aquesta s'haurà d'assignar a l'última línia del cos de la funció (per tal de facilitar l'anàlisi semàntica dels programes).

Les instruccions bàsiques del llenguatge seran assignacions i sentències de control de flux. En particular, el llenguatge disposa d'una estructura condicional (*if-then-else*) i una estructura repetitiva (*while*).

A partir de les operacions mencionades prèviament i l'ús de literals (especificats en bases binària, decimal o hexadecimal), variables i crides a funcions, es construeixen les expressions bàsiques del llenguatge.

Com ja s'ha explicat, aquest llenguatge no simula una memòria i tampoc permet recursivitat. Així, totes les variables a utilitzar han d'haver estat prèviament definides pel programador estàticament. Per tant, l'expressivitat del llenguatge és força limitada (tot i que és més que suficient per a l'objectiu d'aquest projecte).

Finalment, segueixen alguns exemples d'ús del llenguatge. En particular, al Fragment de codi 1 es mostren alguns algorismes elementals utilitzats en teoria de nombres, mentre que al Fragment de codi 2 es mostra una estratègia molt simple per xifrar i desxifrar blocs de dades. En aquests exemples es poden observar les construccions bàsiques que permet el llenguatge.

```
VECTOR_SIZE 64


FUNC mod_int64
    ARG a, b GRA

    IF b < 0 THEN
        b = -b;
    FI;
    WHILE a < 0 DO
        a = a + b;
    ELIHW;
    WHILE a >= b DO
        a = a - b;
    ELIHW;
    mod_int64 = a;
CNUF


FUNC mult_int64
    ARG a, b GRA
    VAR m, s RAV

    m = 0;
    s = 1;
    IF a < 0 THEN s = -s; a = -a; FI;
    IF b < 0 THEN s = -s; b = -b; FI;
    WHILE a != 0 DO
        IF a & 1 THEN
            m = m + b;
        FI;
        a = a >> 1;
        b = b << 1;
    ELIHW;
    IF s == -1 THEN m = -m; FI;
    mult_int64 = m;
CNUF


```

```
40 FUNC div_int64
41     ARG a, b GRA
42     VAR d, sa, sb RAV
43 
44     d = 0;
45     sa = 1;
46     sb = 1;
47     IF a < 0 THEN sa = -1; a = -a; FI;
48     IF b < 0 THEN sb = -1; b = -b; FI;
49     WHILE a >= b DO
50         a = a - b;
51         d = d + 1;
52     ELIHW;
53     IF sa == -1 AND a != 0 THEN
54         d = d + 1;
55     FI;
56     IF sa != sb THEN
57         d = -d;
58     FI;
59     div_int64 = d;
60 CNUF
61 
62 
63 FUNC gcd_int64
64     ARG a, b GRA
65     VAR tmp RAV
66 
67     WHILE b != 0 DO
68         tmp = b;
69         b = mod_int64(a, b);
70         a = tmp;
71     ELIHW;
72     gcd_int64 = a;
73 CNUF
74 
75 
76 FUNC gcd_positive_int64
77     ARG a, b GRA
78 
79     WHILE a != b DO
```

```
        IF a > b THEN
            a = a - b;
        ELSE
            b = b - a;
        FI;
    ELIHW;
    gcd_positive_int64 = a;
CNUF


FUNC jacobi_symbol_int64
    ARG a, b GRA
    VAR r, tmp, end RAV

    end = 0;
    IF b < 0 THEN b = -b; FI;
    WHILE !end DO
        a = mod_int64(a, b);
        WHILE a AND !(a & 1) DO
            tmp = (mult_int64(b, b) - 1) >> 3;
            IF tmp & 1 THEN
                r = -r;
            FI;
            a = a >> 1;
        ELIHW;
        IF a == 1 THEN
            end = 1;
        FI;
        IF gcd_int64(a, b) != 1 THEN
            r = 0;
            end = 1;
        FI;
        tmp = a;
        a = b;
        b = tmp;
    ELIHW;
    jacobi_symbol_int64 = r;
CNUF


```

```
120 FUNC main
121     main = jacobi_symbol_int64(1, 4);
122 CNUF
```

Fragment de codi 1: Exemple d'ús per funcions en enters.

```
1  VECTOR_SIZE 64
2
3
4  FUNC encrypt_int32
5      ARG v, k, kk GRA
6      VAR s, i, d, v0, v1, k0, k1, k2, k3 RAV
7
8      v0 = v & 0x00000000FFFFFFFF;
9      v1 = (v >> 32) & 0xFFFFFFFF00000000;
10     k0 = kk & 0x00000000FFFFFFFF;
11     k1 = (kk >> 32) & 0xFFFFFFFF00000000;
12     k2 = k & 0x00000000FFFFFFFF;
13     k3 = (k >> 32) & 0xFFFFFFFF00000000;
14     d = 0x000000009e3779b9;
15     s = 0;
16     i = 0;
17     WHILE i < 32 DO
18         s = s + d;
19         v0 = v0 + (((v1 << 4) + k0) ^ (v1 + s) ^ ((v1 >> 5) +
               k1));
20         v1 = v1 + (((v0 << 4) + k2) ^ (v0 + s) ^ ((v0 >> 5) +
               k3));
21         i = i + 1;
22     ELIHW;
23     encrypt_int32 = (v0 & 0x00000000FFFFFFFF) | ((v1 << 32) &
           0xFFFFFFFF00000000);
24 CNUF
25
26
27 FUNC decrypt_int32
28     ARG v, k, kk GRA
29     VAR s, i, d, v0, v1, k0, k1, k2, k3 RAV
30
31     v0 = v & 0x00000000FFFFFFFF;
```

```
32      v1 = (v >> 32) & 0xFFFFFFFF00000000;
33      k0 = kk & 0x00000000FFFFFFFF;
34      k1 = (kk >> 32) & 0xFFFFFFFF00000000;
35      k2 = k & 0x00000000FFFFFFFF;
36      k3 = (k >> 32) & 0xFFFFFFFF00000000;
37      d = 0x000000009e3779b9;
38      s = 0x00000000c6ef3720;
39      i = 0;
40      WHILE i < 32 DO
41          v1 = v1 - (((v0 << 4) + k2) ^ (v0 + s) ^ ((v0 >> 5) +
                k3));
42          v0 = v0 - (((v1 << 4) + k0) ^ (v1 + s) ^ ((v1 >> 5) +
                k1));
43          s = s - d;
44          i = i + 1;
45      ELIHW;
46      decrypt_int32 = (v0 & 0x00000000FFFFFFFF) | ((v1 << 32) &
            0xFFFFFFFF00000000);
47  CNUF
48
49
50  FUNC main
51      ARG v, k, kk GRA
52      VAR w, r RAV
53
54      w = encrypt_int32(v, k, kk);
55      IF decrypt_int32(w, k, kk) == v THEN
56          r = 1;
57      ELSE
58          r = 0;
59      FI;
60      main = r;
61  CNUF
```

Fragment de codi 2: Exemple d'ús en un mètode de xifrat.

3 Mòduls del projecte

Dividirem el projecte essencialment en tres mòduls, la funció dels quals es descriu breument a continuació.

3.1 Analitzador lèxico-sintàctic

Com a gairebé qualsevol llenguatge, a la primera part del *pipeline* té lloc l'anàlisi lèxica i sintàctica. Aquest mòdul, generat automàticament amb `ANTLR`, serà el que rebi un fitxer de codi com a entrada i retorni, si el codi resulta ser correcte a nivell sintàctic, el seu AST corresponent. En aquesta secció considerarem que un codi és correcte si aquest es pot obtenir com a resultat de les regles de producció de la gramàtica adjunta al Fragment de codi 3.

3.2 Analitzador semàntic

Un cop generat l'arbre sintàctic, cal realitzar unes certes comprovacions que assegurin que el programa presentat inicialment és totalment correcte i traduïble. Al marge de les condicions habituals presents a la majoria de llenguatges, cal assegurar-se també que no existeix cap successió de crides a funcions que provoquin un cicle. Amb aquesta finalitat, el que es farà serà generar un graf dirigit on cada funció serà un node i, donats dos nodes f i g, només existirà un arc (f, g) si f, en algun punt (o en diversos) del seu codi, crida a g. Considerarem que un programa és correcte respecte de les crides quan el graf generat sigui acíclic. Aquesta restricció imposada elimina, per tant, l'existència de qualsevol tipus de recursivitat (però no les crides multinivell). L'arbre sintàctic generat a la primera fase i el graf de crides derivat d'aquest seran l'entrada que rebrà el mòdul de la següent (i última) fase.

3.3 Generador de *hardware*

Finalment, un cop comprovada la consistència sintàctica i semàntica, es realitza la traducció a *hardware* del programa que s'ha rebut inicialment com a entrada. Aquesta traducció consisteix a convertir el codi en `Verilog` sintetitzable el qual descrigui un circuit que implementi el programa inicial proposat. De manera similar a com fan molts llenguatges compilats, aquesta traducció consistirà en l'aplicació sistemàtica de regles predefinides que indiquen com s'ha d'implementar cadascuna de les construccions del llenguatge. Cal especificar que, en aquesta

fase, el graf de crides prèviament generat per l'analitzador semàntic és especialment útil, ja que aquest ens indicarà quins mòduls de funcions s'han d'interconnectar.

4 Distribució del treball

Cal mencionar que la part lèxico-sintàctica s'ha dissenyat, desenvolupat i depurat entre els dos integrants del grup. És per aquest motiu que aquesta part (la qual correspondria al primer mòdul dels presentats) no apareix a la repartició de feina que presentem a continuació:

- Sergio Rodríguez: Part semàntica, generació del graf de crides. Aquesta part correspondria al segon mòdul dels enumerats anteriorment.
- Francesc Gispert: Generació del *hardware* a partir del codi. És a dir, el tercer i últim mòdul llistat.

També es vol fer notar que aquesta repartició de feina és només orientativa i pot variar a la pràctica. En particular, aquesta distribució només reflecteix qui assumirà més responsabilitats en cadascun dels dos mòduls restants, però els dos integrants del grup col·laborarem activament en tot el desenvolupament del projecte.

5 Gramàtica del llenguatge

Tot seguit es mostra la gramàtica del llenguatge dissenyat, utilitzant ja el llenguatge ANTLR per a descriure-la.

```
grammar hw_compilation;


options {
    output = AST;
}

tokens {
    ROOT;
}


@header {
    package parser;
}

@lexer::header {
    package parser;
}


program : VECTOR_SIZE int_value (function)+ EOF -> ^(ROOT
   int_value (function)+)
        ;

function: FUNC^ ID args? vars? list_instructions CNUF!
        ;

args    : ARG^ ID (COMMA! ID)* GRA!
        ;

vars    : VAR^ ID (COMMA! ID)* RAV!
        ;

list_instructions : (instruction SEPARATOR!)+
```

```
35          ;
36 instruction : assignment_stmt
37            | ite_stmt
38            | while_stmt
39            ;
40
41 expr     : andterm (OR^ andterm)*;
42 andterm  : bwor (AND^ bwor)*;
43 bwor     : bwxor (BWOR^ bwxor)*;
44 bwxor    : bwand (BWXOR^ bwand)*;
45 bwand    : cmpterm (BWAND^ cmpterm)*;
46 cmpterm  : relational_term ((LEQ^|NEQ^) relational_term)*;
47 relational_term : bwshift ((LT^|LTE^|GT^|GTE^) bwshift)*;
48 bwshift : arith_term ((SHIFT_LEFT^ | SHIFT_RIGHT^) arith_term)
      *;
49 arith_term : unary_term ((PLUS^|MINUS^) unary_term)*;
50 unary_term : (BWNOT^|LNOT^|PLUS^|MINUS^)? atom;
51 atom     : int_value
52          | LPAREN! expr RPAREN!
53          | func_call
54          | ID
55          ;
56
57 assignment_stmt : ID EQ^ expr
58            ;
59
60 ite_stmt : IF^ expr THEN! list_instructions else_stmt? FI!
61           ;
62
63 else_stmt: ELSE^ list_instructions
64           ;
65
66 while_stmt : WHILE^ expr DO! list_instructions ELIHW!
67             ;
68
69 func_call : ID^ LPAREN! (expr (COMMA! expr)*)? RPAREN!
70            ;
71
72 int_value : BINARY | DEC | HEX
73            ;
```

```
74
75
76  VECTOR_SIZE : 'VECTOR_SIZE';
77  FUNC : 'FUNC' ;
78  CNUF : 'CNUF' ;
79  ARG  : 'ARG'  ;
80  GRA  : 'GRA'  ;
81  VAR  : 'VAR'  ;
82  RAV  : 'RAV'  ;
83  IF   : 'IF'   ;
84  THEN : 'THEN' ;
85  ELSE : 'ELSE' ;
86  FI   : 'FI'   ;
87  WHILE: 'WHILE';
88  DO   : 'DO'   ;
89  ELIHW: 'ELIHW';
90  OR: 'OR'      ;
91  AND: 'AND'    ;
92  BWOR: '|'     ;
93  BWXOR: '^'    ;
94  BWAND: '&'    ;
95  SHIFT_LEFT: '<<';
96  SHIFT_RIGHT: '>>';
97  LEQ: '==';
98  NEQ: '!=';
99  LT: '<';
100 LTE: '<=';
101 GT: '>';
102 GTE: '>=';
103 PLUS: '+';
104 MINUS: '-';
105 BWNOT: '~';
106 LNOT: '!';
107 LPAREN: '('   ;
108 RPAREN: ')'   ;
109 EQ   : '='    ;
110 COMMA: ','    ;
111 SEPARATOR: ';';
112 BINARY : '0b' ('0'..'1')+ ;
113 HEX : '0x' ('0'..'9' | 'a'..'f' | 'A'..'F')+ ;
```

```
114 DEC : ('0'..'9')+ ;
115 ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;
116 // White spaces
117 WS      : ( ' '
118         | '\t'
119         | '\r'
120         | '\n'
121         ) {$channel=HIDDEN;}
122         ;
```

Fragment de codi 3: Gramàtica del llenguatge.

Referències

[1] Wikipedia. *Jacobi symbol — Wikipedia, The Free Encyclopedia*. 2015. URL: http://en.wikipedia.org/w/index.php?title=Jacobi_symbol&oldid=649400113 (cons. 19-3-2015).

[2] Wikipedia. *Tiny Encryption Algorithm — Wikipedia, The Free Encyclopedia*. 2015. URL: http://en.wikipedia.org/w/index.php?title=Tiny_Encryption_Algorithm&oldid=646326324 (cons. 19-3-2015).

[3] Wirth, Niklaus. "Hardware compilation. Translating programs into circuits". A: *Computer* 31.6 (1998), pàg. 25 - 31.