Compiladors
Documentació de HW–Compilation

Gispert Sánchez, Francesc-Xavier Rodríguez Guasch, Sergio

5 de juny 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 construïm un traductor que, a partir dels programes especificats en aquest llenguatge, genera automàticament els circuits corresponents. En particular, aquest llenguatge de programació tracta 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 disposa 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, és la falta de memòria (cosa que, d'altra banda, facilita 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 és molt simple i limitat. Això ens permet concentrarnos 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 és, com el seu nom indica, el punt d'entrada a l'execució. Cadascuna d'aquestes funcions, excepte `main`, 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 facilita el procés de traducció, ja que, en traduir el programa a un circuit, cada variable correspon 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 emmagatzema el valor de retorn: aquesta s'ha d'assignar a l'última línia del cos de la funció (per tal de facilitar l'anàlisi semàntica dels programes) i, a diferència de les altres variables i paràmetres, no es pot utilitzar en expressions dins de la pròpia funció.

Les instruccions bàsiques del llenguatge són 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).

3 Mòduls del projecte

El projecte s'ha dividit essencialment en tres mòduls, la funció dels quals es descriu breument a continuació. Així mateix, en seccions posteriors es descriuen més detalladament algunes parts del projecte; concretament, aquelles parts que hem considerat més importants per a la correcta comprensió del projecte.

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 1.

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 fa és generar un graf dirigit on cada funció és un node i, donats dos nodes f i g, només existeix 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 són l'entrada que rep 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 que descrigui un circuit que implementi el programa inicial proposat. De manera similar a com fan molts llenguatges compilats, aquesta traducció consisteix 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 indica quins mòduls de funcions s'han d'interconnectar.

4 Operacions sobre l'arbre sintàctic

En aquesta secció, s'ofereix una explicació més detallada de la obtenció de certa informació intermèdia que s'ha encabit en el mòdul d'anàlisi semàntica.

Cal recordar que aquesta part és la que s'executa immediatament després de l'anàlisi lèxico-sintàctica (si aquesta ha resultat reeixida). En aquesta part es comprova que les restriccions més típiques de llenguatges similars a l'emprat se satisfacin (per exemple, que no s'utilitzin variables no declarades, que no es redeclarin variables i altres normes similars). Donades les particularitats del llenguatge, també cal comprovar que cap de totes les possibles successions de crides que poden tenir lloc al programa desemboquin en recursivitat.

Aquesta part d'anàlisi semàntica també s'encarrega de realitzar uns certs càlculs necessaris per a la part posterior, la qual s'encarrega de traduir finalment el programa donat en un circuit descrit en `Verilog` sintetitzable. També es construeix, de forma simultània a les comprovacions semàntiques i els càlculs anteriorment esmentats, un graf G i el seu transposat G^T, el qual conté com a nodes les funcions del programa. Una aresta (f, g) és present a G si f crida a g en algun punt del programa (o en diversos).

La tasca d'anàlisi semàntica s'ha dividit en tres etapes: preprocessament, recorregut explícit de l'AST generat al mòdul anterior i comprovació de funcions no utilitzades i recursivitat. Cal remarcar que, al disseny presentat, les comprovacions semàntiques i els càlculs necessaris per a la següent fase es duen a terme de forma simultània, ja que aquests dos procediments no presenten cap mena de dependència entre ells.

4.1 Preprocessament

És útil obtenir certa informació prèviament al recorregut de l'AST obtingut a la fase anterior. Aquesta tasca consisteix a obtenir els noms i les referències als nodes de les funcions declarades al programa. Amb aquestes dades es pot comprovar que cap nom de funció es declari més d'una vegada i que hi hagi una funció principal definida. Aquestes dades també permeten inicialitzar estructures de dades auxiliars, com ara els dos grafs de crides (els quals consistiran, després d'aquesta fase, en grafs amb tants nodes com funcions hi hagi i sense cap aresta). Si es troba qualsevol error, l'execució s'avorta en aquesta fase i no es passa a la següent.

4.2 Recorregut de l'AST

En aquesta etapa de l'anàlisi es recorre l'arbre sintàctic pròpiament dit. Donades les necessitats específiques del llenguatge, la classe que representa l'arbre sintàctic no és la pròpia d'ANTLR, sinó una extensió d'aquesta. Concretament, s'ha estès per permetre que cada node contingui informació sobre quina és la seva funció d'origen. Pels nodes que representen una crida a funció també es guarda un enter que indica quin número de crida representen dins la seva funció origen (és a dir, es tracta d'identificadors de les crides: si l'identificador d'una crida és 5, vol dir que aquest node representa la cinquena crida a funció que es pot trobar al codi de la seva funció origen). Quelcom similar es fa per a les assignacions.

L'estratègia de recorregut és senzilla: s'obté el node que representa la funció principal i es recorre tot l'arbre accessible des d'aquest. Durant el recorregut, es pot estar en diferents estats segons els tipus de node: funció, llista d'instruccions, instrucció i expressió. A cada estat es comproven diverses restriccions semàntiques i es construeixen diferents parts de les estructures de dades auxiliars anteriorment comentades.

En els nodes que representen funcions es comprova, en primer lloc, que el node en qüestió no hagi estat processat prèviament. Tot seguit, es comprova que els noms de les variables i paràmetres definits siguin consistents (tant entre ells com amb els noms d'altres funcions). Per consistència hem d'entendre que no hi hagi dos elements amb el mateix nom. Aprofitem per recordar que l'àmbit de visibilitat pel que fa a variables i paràmetres és únicament local i que, per tant, dues funcions poden tenir variables i paràmetres amb els mateixos noms sense cap problema. Si es troba algun error en aquesta part, s'avorta l'anàlisi sense passar a la següent fase. També s'aprofita per a inicialitzar localment unes estructures de dades que mantenen el comptatge d'assignacions realitzades a les variables o paràmetres d'aquesta funció, les quals seran necessàries per a poder assignar l'identificador d'assignacions anteriorment esmentat. Finalment, es comprova que l'última instrucció contingui una assignació la part esquerra de la qual sigui el propi nom de la funció, ja que això, en aquest llenguatge, es considera el valor de retorn. Donat que no existeixen efectes laterals, es pot considerar un error el fet de tenir una funció sense un valor de retorn correctament definit. Si totes les comprovacions han resultat correctes, es procedeix a recórrer la llista d'instruccions, la qual representa el codi de la funció. En l'anàlisi del cos de la funció, doncs, es manté un registre del nom de la funció d'origen i els noms visibles (arguments i variables).

Per a les llistes d'instruccions, simplement es processen tots els fills (els quals

seran instruccions) de manera successiva, parant si algun d'aquests provoca un error.

Els nodes que representen instruccions es tracten per casos (un cas per tipus d'instrucció). Els casos possibles són: assignació, condicional (*if*) i bucle (*while*).

Pel cas de l'assignació, es comprova, en primer lloc, que l'identificador de la part esquerra sigui, en efecte, una variable visible dins l'àmbit de la funció a la qual s'està. Tot seguit, s'assigna un identificador d'assignació al node en qüestió i s'actualitza el comptatge d'aquestes. Finalment, si no hi ha hagut errors, es recorre l'expressió que correspon a la part dreta de l'assignació.

En els condicionals, es comença recorrent l'expressió que correspon a la condició. Si no hi ha hagut cap error, es recorre la llista d'instruccions del *then*. Si no hi ha hagut cap error, es comprova si existeix un últim node del tipus llista d'instruccions (el qual representaria l'*else*); en cas afirmatiu, també es recorre. Els bucles es processen de forma similar, obviant la part de l'*else*.

L'últim cas a tenir en compte són els nodes que corresponen a les expressions. Aquests nodes també es tracten diferent segons de quin tipus siguin. Aquí podem distingir els següents casos: identificador, literal enter, crida a funció i operadors aritmètics.

Pel cas de l'identificador, simplement es comprova que aquest faci referència a algun dels noms de variables o paràmetres visibles en l'àmbit en qüestió.

Els literals s'ignoren i no es fa res amb ells. Això es degut a què, en aquesta part, no es necessita comprovar res dels seus valors: els problemes que aquests podrien donar en temes de correcció es detecten a la fase d'anàlisi sintàctica, la qual és anterior a aquesta.

Les crides a funcions es tracten com segueix: es comprova primer que la funció cridada existeix i que l'aritat proposada (el nombre d'arguments) coincideix amb l'aritat real d'aquesta. També es recorren les expressions que representen els paràmetres. Finalment, es recorre la funció que s'està cridant. Addicionalment, s'afegeixen les arestes corresponents als dos grafs de crides.

Finalment, pels operadors aritmètics o lògics es comproven recursivament les expressions que es tenen com a operands.

4.3 Comprovació de recursivitat

Un cop acabat el processament descrit a la secció anterior, es procedeix a comprovar que no existeixi cap successió de crides que desemboqui en recursivitat, ja sigui directa o indirecta. Per això, es mira que cap funció no es cridi a si mateixa i que el graf dirigit de crides G generat no contingui cap component fortament con-

nexa de més d'un vèrtex. Donat que l'algorisme fet servir per aquesta detecció és constructiu, es pot donar com a missatge d'error el conjunt (o conjunts) exacte de funcions que creen recursivitat. Com a efecte lateral de l'execució d'aquest algorisme, també s'obté una col·lecció de noms de funcions, els quals representen les funcions que són potencialment utilitzades en l'execució del programa. Això és especialment útil per a la següent fase, ja que permet fer una traducció selectiva, podent resultar en programes més petits que els que s'obtindrien si la traducció fos del tot sistemàtica. Aquesta col·lecció també es fa servir per emetre avisos de codi no emprat al programa final.

5 Traducció a circuits

En aquesta secció, es descriuen les regles dissenyades per traduir de forma automàtica el codi del nostre llenguatge a circuits lògics (expressats en el llenguatge `Verilog`). Les idees fonamentals estan basades en l'exposició de [5] (tot i que s'han ampliat per permetre construccions diferents de les mostrades en l'article citat).

Les explicacions d'aquesta secció es basen en diversos diagrames de circuits que en faciliten la comprensió. En aquests, s'ha seguit la convenció d'utilitzar línies fines per als cables d'un únic bit i línies més gruixudes per representar els busos de n bits (on n és un nombre especificat per l'usuari del llenguatge al principi del programa a traduir; en els subsegüents diagrames s'utilitzarà $n = 2$ per simplicitat). A més, aquests circuits corresponen a l'execució d'un codi i, per tant, s'ha intentat que es puguin interpretar d'aquesta manera veient-los "d'esquerra a dreta" (en particular, quan apareixen diversos senyals amb el mateix nom, el senyal de l'esquerra correspon a l'entrada d'aquell fragment de circuit, mentre que el senyal de la dreta correspon a la sortida).

Els circuits s'expressen utilitzant només portes lògiques i biestables de dos tipus. Els del primer tipus tenen un senyal d'entrada D, un de sortida Q i un de reinici R. Quan hi ha un flanc ascendent del rellotge es canvia el valor de Q: si el valor de R és 1, aleshores Q passa a valdre 0; altrament, el valor de Q passa a ser el valor de D. Els biestables del segon tipus funcionen de forma similar però, a més, tenen un senyal d'activació E. Així, quan hi ha un flanc ascendent de rellotge i el valor de R és 0, només es canvia el valor de Q pel de D si el valor de E és 1 (altrament, el valor de Q roman inalterat).

Per traduir un programa a *hardware*, es van generant recursivament fragments d'un circuit lògic seguint l'estructura de l'AST que representa les instruccions del programa. Aquests circuits es componen de dues parts: d'una banda, hi ha una part principal del circuit que efectua els càlculs expressats en les instruccions del programa; de l'altra, hi ha una segona part igual d'important que controla el flux de l'execució (és a dir, activa i desactiva els diferents components del primer circuit per garantir que les operacions s'efectuen seguint l'ordre correcte). Vegem ara com es tradueixen els diferents tipus d'instruccions.

Per traduir un bloc d'instruccions consecutives, senzillament cal traduir cadascuna de les instruccions per separat i encadenar els senyals d'activació. D'aquesta manera, es fa que el senyal de sortida del circuit corresponent a la primera instrucció sigui el senyal d'entrada del circuit corresponent a la segona instrucció, i així successivament fins a l'última. La Figura 1 mostra la forma dels circuits

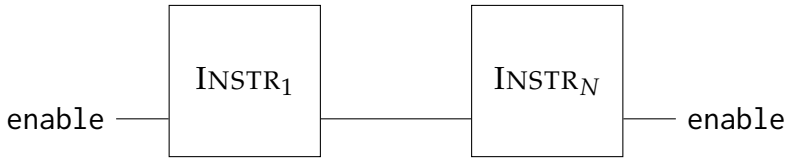

Figura 1: Circuit corresponent a una llista de N instruccions (en el diagrama, per simplicitat, $N = 2$) consecutives.

obtinguts d'aquesta forma.

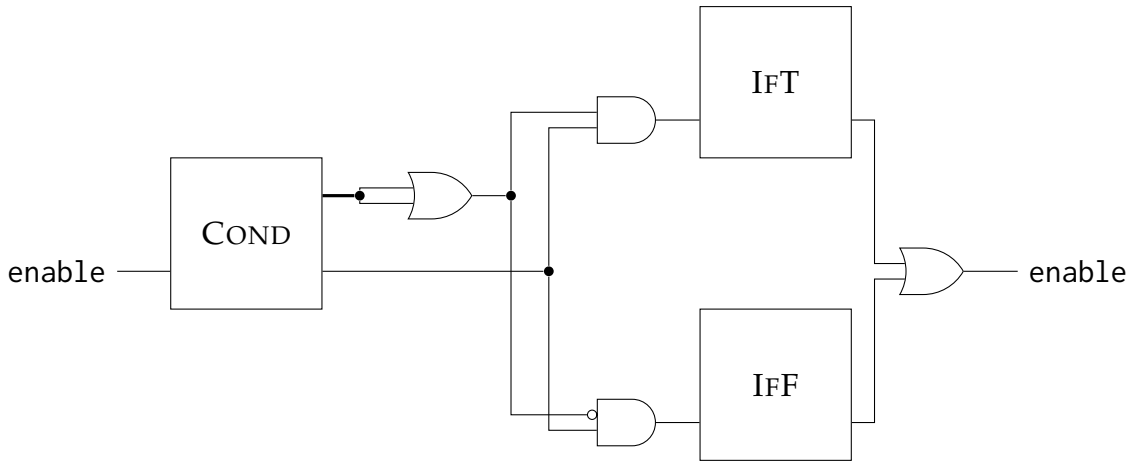

Figura 2: Circuit corresponent a una sentència condicional.

La Figura 2 mostra el circuit generat per a una sentència condicional. En qualsevol dels casos, primerament cal avaluar la condició i, per tant, es genera recursivament el circuit COND corresponent i s'hi col·loca el senyal d'activació de la sentència condicional directament. Com que el llenguatge dissenyat treballa amb vectors de n bits fins i tot per a les condicions, cal utilitzar una porta *or* amb entrades cadascun d'aquests n bits per comprovar si la condició es verifica o no (en aquest llenguatge, l'únic valor amb el valor lògic fals és el 0). Els dos circuits corresponents a les branques condicional i alternativa d'aquesta instrucció, IFT i IFF, també es generen recursivament i cal muntar un circuit d'activació que n'activi exactament un dels dos (en funció del valor de la condició avaluada). A més, aquests circuits s'han d'activar justament després que el circuit d'avaluació de la condició. Així, doncs, el senyal d'activació del circuit IFT es connecta amb el resultat d'una porta *and* amb entrades la condició avaluada i el senyal d'activació de sortida de COND; anàlogament, el senyal d'activació del circuit IFF es connecta amb el resultat d'una porta *and* amb entrades la condició avaluada negada i el senyal d'activació del circuit de sortida de COND. Finalment, el senyal d'activació resultant després d'aquesta instrucció s'ha d'obtenir mitjançant una porta *or*

amb els senyals d'activació que surten de IFT i IFF (o sigui, la instrucció s'acaba d'executar quan alguna de les dues branques condicionals finalitza la seva execució). En cas que no hi hagi una branca alternativa (és a dir, un *else*), el circuit IFF desapareix i la resta del circuit queda exactament igual.

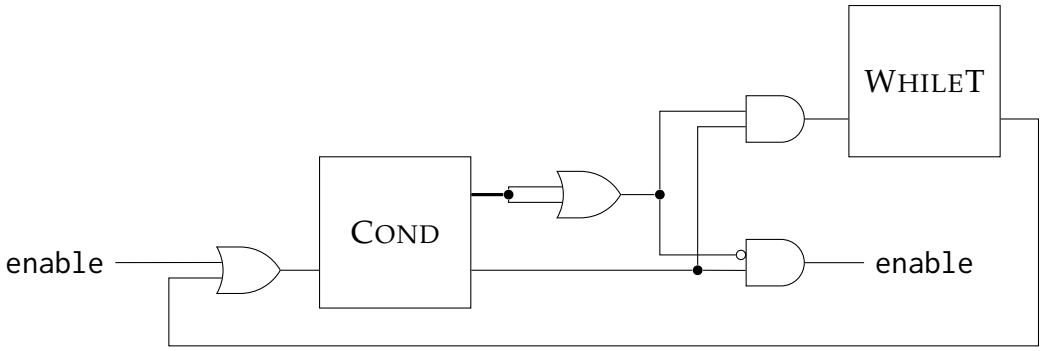

Figura 3: Circuit corresponent a una sentència repetitiva.

El circuit generat per a una sentència repetitiva s'il·lustra a la Figura 3. El fragment COND d'aquest (que correspon a l'avaluació de la condició) i l'activació de WHILET (corresponent al bloc d'instruccions que conforma el cos de la sentència repetitiva) en funció del resultat de la condició és igual que el d'una sentència condicional (sense *else*) i, per tant, n'ometem una descripció en detall. En aquest cas, però, el senyal d'activació per al circuit COND es calcula de forma diferent, ja que s'ha de poder repetir després de cada iteració. D'una banda, aquest s'ha d'activar quan s'inicia l'execució de la instrucció (abans de començar cap iteració); d'altra banda, també cal activar l'execució de la condició just després d'executar una iteració del cos de la sentència repetitiva. Per tant, es col·loca una porta *or* que té per entrades el senyal d'activació d'entrada de la instrucció i el senyal de sortida del circuit WHILET i es connecta la sortida d'aquesta porta al senyal d'activació d'entrada del circuit COND.

Fins al moment ja s'ha explicat el mètode seguit per a la generació dels circuits que controlen el flux d'execució del programa. Així, doncs, queda descriure els circuits que es generen per a definir les variables i les funcions del programa (en aquest document no s'explica com dissenyar un circuit per a executar les operacions aritmètiques i lògiques bàsiques, ja que no és l'objectiu del projecte; de fet, aquesta tasca es deixa a l'eina de síntesi o de simulació del circuit perquè les operacions s'expressen directament en la sintaxi de `Verilog`).

La Figura 4 mostra el circuit utilitzat per a definir una variable a la qual s'assigna algun valor en N punts diferents del programa (aquí, entenem variable en un sentit ampli: un paràmetre d'una funció es tracta de la mateixa manera que una variable i, en aquest cas, en les crides a una funció es fa una assignació per

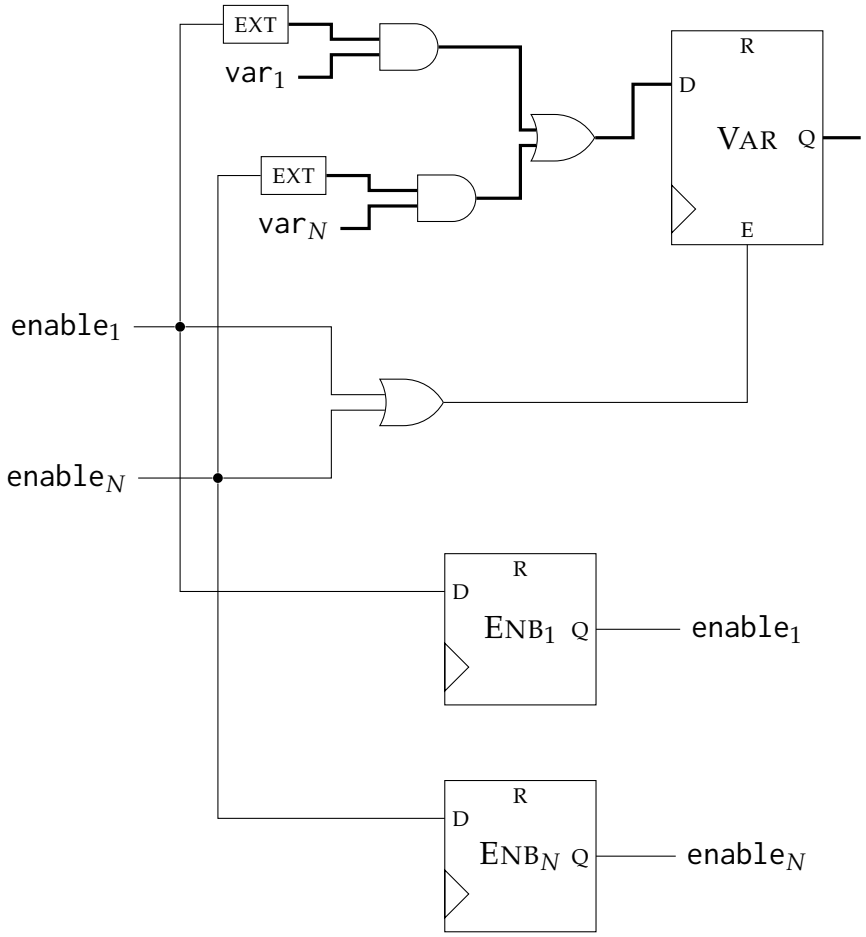

Figura 4: Circuit generat per a una variable amb N assignacions (en el diagrama, per simplicitat, $N = 2$).

cadascun dels arguments). Per a emmagatzemar el valor de la variable, s'utilitza un biestable VAR amb senyal d'activació. El biestable ha de canviar de valor cada vegada que es realitza alguna de les assignacions, de manera que el senyal d'activació E d'aquest està connectat a la sortida d'una porta *or* que té per entrades tots els senyals d'activació de les assignacions. En aquest cas, el senyal d'entrada D ha de filtrar el valor corresponent a l'assignació que s'està executant. Per fer-ho, es repliquen els senyals d'activació fins a obtenir busos de n bits i aquests s'utilitzen com a entrades de portes *and* juntament amb els valors a assignar (així que la sortida de la porta corresponent a l'assignació en execució és el valor a assignar i les sortides de totes les altres portes són 0); finalment, les sortides d'aquestes portes *and* s'utilitzen com a entrada d'una porta *or* de la qual s'obté el valor a assignar. D'altra banda, una assignació tarda un cicle de rellotge a tenir lloc i, per tant, cal retardar un cicle la propagació dels senyals d'activació de les assignacions a VAR mitjançant biestables $\text{ENB}_1, \ldots, \text{ENB}_N$ sense senyal d'activació.

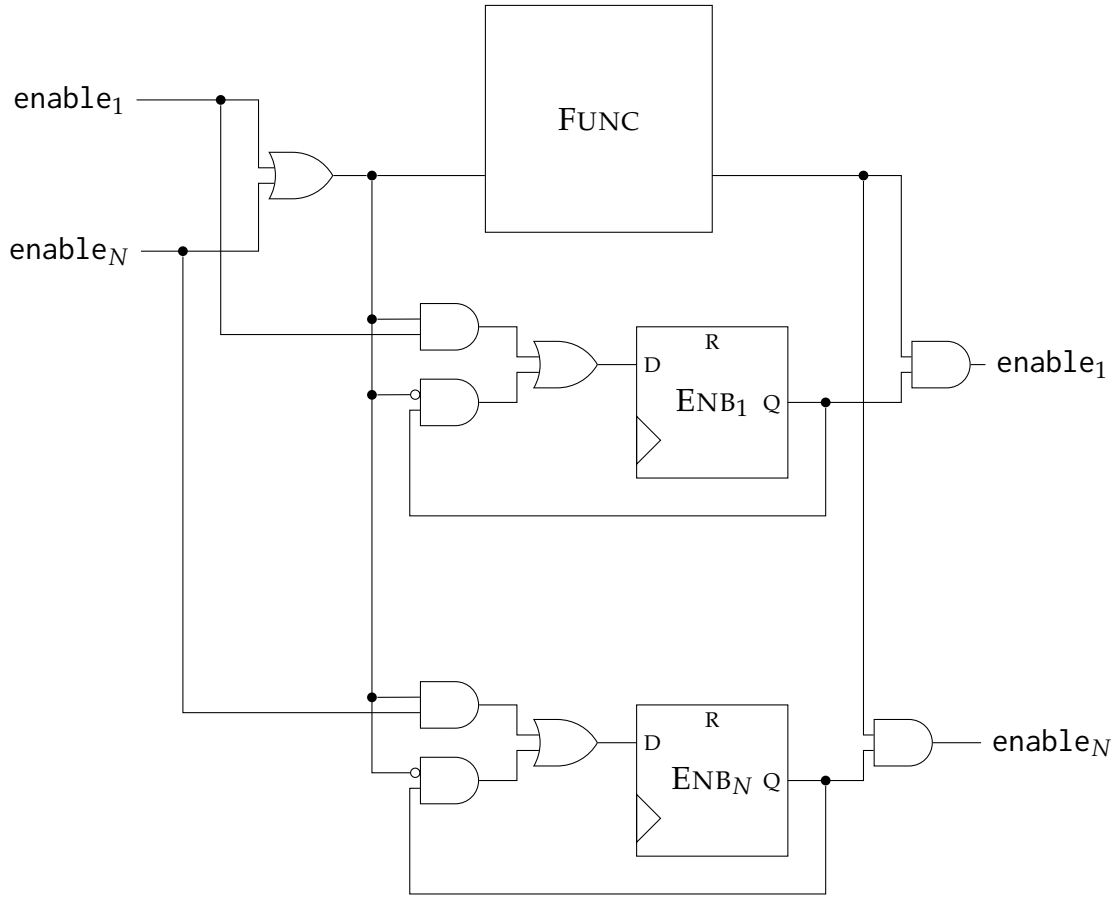

Figura 5: Circuit generat per a una funció amb N crides (en el diagrama, per simplicitat, $N = 2$).

Per a definir una funció tenint en compte les crides que s'hi fan també cal construir un circuit similar, tal com s'observa a la Figura 5. En particular, en aquest diagrama es considera una funció a la qual es fan crides en N punts diferents del programa. Primer de tot, es tradueix recursivament el cos de la funció per generar el circuit FUNC. Aquest s'ha d'activar quan hi hagi alguna crida a la funció; és a dir, el senyal d'activació d'entrada és el resultat d'una porta *or* amb entrades els senyals d'activació de cadascuna de les crides. Ara bé, en acabar l'execució de la funció, cal saber quina crida s'estava executant per poder retornar al punt del programa adequat. Per aconseguir-ho, per a cada crida es col·loca un biestable sense senyal d'activació juntament amb un circuit que permet emmagatzemar el valor del senyal d'activació d'entrada durant l'execució de la funció. Concretament, per a cada crida donada, s'utilitzen dues portes *and*: una amb entrades el senyal d'activació d'entrada del cos de la funció i el senyal d'activació de la crida (així se sap si aquella crida s'ha activat), i una altra amb entrades el senyal d'activació d'entrada del cos de la funció negat i el senyal Q del biestable corresponent

(així es manté el valor previ mentre s'executa la funció). Les sortides d'aquestes dues es combinen amb una porta *or* i el resultat d'aquesta es connecta al senyal D del biestable (és a dir, quan comença una crida, el biestable corresponent a la crida pren el valor 1 i tots els altres biestables prenen el valor 0, i aquests valors es mantenen durant tota l'execució de la funció i, de fet, fins a la següent crida). Finalment, els senyals d'activació de sortida es calculen amb una porta *and* amb entrades el senyal Q del biestable corresponent i el senyal d'activació de sortida del cos de la funció (d'aquesta manera, quan l'execució del cos de la funció acaba, es retorna al punt on s'ha cridat i no a cap altre). Com que l'execució d'una funció triga almenys un cicle de rellotge (per l'assignació final obligatòria), aquest circuit funciona correctament.

Les crides a funcions es redueixen als dos casos anteriors. És a dir, una crida es descompon en dues parts: en la primera, els arguments s'assignen seqüencialment a les variables de la funció cridada que actuen com a paràmetres (per tant, tan sols cal connectar els valors dels arguments i els senyals d'activació al circuit explicat per a les assignacions); en la segona, s'activa l'execució de la funció cridada (per tant, tan sols cal connectar els senyals d'activació al circuit explicat per a la definició de funcions). Per utilitzar el valor retornat per la crida, tan sols cal utilitzar la variable especial amb el mateix nom de la funció (o sigui, es pot connectar directament el senyal Q del biestable que emmagatzema aquesta variable al circuit del càlcul de l'expressió on s'ha fet la crida).

Ara que s'ha descrit formalment el mètode de generació de circuits, passem a comentar breument alguns detalls particulars del programa que s'encarrega de fer-ho. Aquest parteix des de l'arrel de l'AST i, per a cada funció, genera el circuit corresponent a totes les variables (incloent paràmetres i la variable especial de retorn) d'aquesta i llavors genera el circuit que controla les crides a aquesta funció. Seguidament, es tradueix recursivament el cos de la funció segons les regles explicades i seguint els nodes de l'arbre. Per connectar correctament els diferents components, quan s'accedeix a una funció es coneixen el nombre de crides i el nombre d'assignacions a cadascuna de les seves variables (gràcies al procés explicat en la Secció 4); així, es poden generar tants cables per a les assignacions a variables i les crides a la funció com faci falta. D'altra banda, quan, en la traducció del cos d'una funció, s'arriba a una assignació o a una crida, es disposa d'un identificador numèric que permet associar-lo unívocament a un dels cables definits prèviament.

Llevat d'alguns punts més importants del circuit (com ara els descrits), la majoria de cables es defineixen en el codi en `Verilog` amb noms formats per un

nombre per evitar repeticions (però no tenen un significat explicatiu). El motiu és que, en la generació automàtica de circuits, per a cada instrucció del codi original cal definir una multitud de variables en `Verilog`.

El programa que hem desenvolupat, doncs, segueix el mètode descrit per generar un mòdul en `Verilog` amb paràmetres d'entrada el senyal de rellotge, un senyal de reinici dels biestables (connectat al seu port `R`) i el senyal d'activació d'entrada i paràmetres de sortida el senyal d'activació de sortida i el resultat de la funció `main`. Per tant, es pot simular amb alguna eina com `Icarus Verilog` creant un *testbench* en què es generi un senyal de rellotge i s'instanciï el mòdul generat. Cal posar el senyal de reinici a 1 durant un cicle al principi i, un cop posats a 0 tots els biestables, activar l'execució posant a 1 el senyal d'activació durant un cicle. El flanc ascendent en el senyal d'activació de sortida indica que l'execució ha finalitzat correctament i es disposa del resultat.

6 Distribució del treball

La part lèxico-sintàctica d'aquest projecte s'ha dissenyat, desenvolupat i depurat conjuntament entre els dos integrants del grup, ja que aquesta requereix la presa de moltes decisions importants. Aquesta part correspon al primer mòdul explicat. La resta del projecte s'ha dividit essencialment de la següent manera:

- 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.

Aquesta repartició, tanmateix, no s'ha seguit estrictament en tot el desenvolupament del projecte. En particular, en les fases d'especificació inicial, de depuració del codi i de construcció de jocs de prova, així com en la presa de decisions importants, ambdós membres del grup hem col·laborat activament. És a dir, aquesta distribució reflecteix qui ha assumit més responsabilitats en cadascun dels dos mòduls finals, però els dos integrants del grup hem col·laborat en tot el desenvolupament del projecte.

7 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;
    INSTR_LIST;
    FUNC_CALL;
    PARAM;
}


@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!
```

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

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

```
114 LPAREN: '(' ;
115 RPAREN: ')' ;
116 EQ : '=' ;
117 COMMA: ',' ;
118 SEPARATOR: ';';
119 BINARY : '0b' ('0'..'1')+ ;
120 HEX : '0x' ('0'..'9' | 'a'..'f' | 'A'..'F')+ ;
121 DEC : ('0'..'9')+ ;
122 ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
123 // White spaces
124 WS  : ( ' '
125        | '\t'
126        | '\r'
127        | '\n'
128        ) {$channel=HIDDEN;}
129        ;
```

Fragment de codi 1: Gramàtica del llenguatge.

8 Exemples d'ús

Finalment, segueixen alguns exemples no trivials d'ús del llenguatge. En particular, al Fragment de codi 2 es mostren alguns algorismes elementals utilitzats en teoria de nombres, mentre que al Fragment de codi 3 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.

D'altra banda, amb el codi adjunt a aquest document, s'inclouen exemples que realitzen càlculs sense gaire sentit pràctic (o fins i tot incorrectes) i que han estat utilitzats durant el desenvolupament del projecte per comprovar el correcte funcionament de les diferents funcionalitats implementades.

```
1  VECTOR_SIZE 64
2
3
4  FUNC mod_int64
5      ARG a, b GRA
6
7      IF b < 0 THEN
8          b = -b;
9      FI;
10     WHILE a < 0 DO
11         a = a + b;
12     ELIHW;
13     WHILE a >= b DO
14         a = a - b;
15     ELIHW;
16     mod_int64 = a;
17 CNUF
18
19
20 FUNC mult_int64
21     ARG a, b GRA
22     VAR m, s RAV
23
24     m = 0;
25     s = 1;
26     IF a < 0 THEN s = -s; a = -a; FI;
27     IF b < 0 THEN s = -s; b = -b; FI;
```

```
28      WHILE a != 0 DO
29          IF a & 1 THEN
30              m = m + b;
31          FI;
32          a = a >> 1;
33          b = b << 1;
34      ELIHW;
35      IF s == -1 THEN m = -m; FI;
36      mult_int64 = m;
37  CNUF
38
39
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
80          IF a > b THEN
81              a = a - b;
82          ELSE
83              b = b - a;
84          FI;
85      ELIHW;
86      gcd_positive_int64 = a;
87  CNUF
88
89
90  FUNC jacobi_symbol_int64
91      ARG a, b GRA
92      VAR r, tmp, end RAV
93
94      r = 1;
95      end = 0;
96      IF b < 0 THEN b = -b; FI;
97      WHILE !end DO
98          a = mod_int64(a, b);
99          WHILE a AND !(a & 1) DO
100             tmp = (mult_int64(b, b) - 1) >> 3;
101             IF tmp & 1 THEN
102                 r = -r;
103             FI;
104             a = a >> 1;
105         ELIHW;
106         IF a == 1 THEN
107             end = 1;
```

```
        ELSE
            IF gcd_int64(a, b) != 1 THEN
                r = 0;
                end = 1;
            ELSE
                IF mod_int64(a, 4) == 3 AND mod_int64(b, 4) == 3 THEN
                    r = -r;
                FI;
                tmp = a;
                a = b;
                b = tmp;
            FI;
        FI;
    ELIHW;
    jacobi_symbol_int64 = r;
CNUF


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

Fragment de codi 2: Exemple d'ús per funcions en enters.

```
VECTOR_SIZE 64


FUNC encrypt_int32
    ARG v, k, kk GRA
    VAR s, i, d, v0, v1, k0, k1, k2, k3 RAV

    v0 = v & 0x00000000FFFFFFFF;
    v1 = (v >> 32) & 0x00000000FFFFFFFF;
    k0 = kk & 0x00000000FFFFFFFF;
    k1 = (kk >> 32) & 0x00000000FFFFFFFF;
    k2 = k & 0x00000000FFFFFFFF;
    k3 = (k >> 32) & 0x00000000FFFFFFFF;
    d = 0x000000009e3779b9;
    s = 0;
    i = 0;
```

```
17      WHILE i < 32 DO
18          s = s + d;
19          v0 = (v0 + (((v1 << 4) + k0) ^ (v1 + s) ^ ((v1 >> 5) + k1))) &
                0x00000000FFFFFFFF;
20          v1 = (v1 + (((v0 << 4) + k2) ^ (v0 + s) ^ ((v0 >> 5) + k3))) &
                0x00000000FFFFFFFF;
21          i = i + 1;
22      ELIHW;
23      encrypt_int32 = v0 | (v1 << 32);
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) & 0x00000000FFFFFFFF;
33      k0 = kk & 0x00000000FFFFFFFF;
34      k1 = (kk >> 32) & 0x00000000FFFFFFFF;
35      k2 = k & 0x00000000FFFFFFFF;
36      k3 = (k >> 32) & 0x00000000FFFFFFFF;
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))) &
                0x00000000FFFFFFFF;
42          v0 = (v0 - (((v1 << 4) + k0) ^ (v1 + s) ^ ((v1 >> 5) + k1))) &
                0x00000000FFFFFFFF;
43          s = s - d;
44          i = i + 1;
45      ELIHW;
46      decrypt_int32 = v0 | (v1 << 32);
47  CNUF
48
49
50  FUNC main
51      VAR v, k, kk, w, r RAV
52
```

```
53      k=0x123456789abcdef;
54      kk=0xfedcba987654321;
55      v = 314;
56
57      w = encrypt_int32(v, k, kk);
58      IF decrypt_int32(w, k, kk) == v THEN
59          r = 1;
60      ELSE
61          r = 0;
62      FI;
63      main = r;
64  CNUF
```

Fragment de codi 3: Exemple d'ús en un mètode de xifrat.

Referències

[1] Icarus Verilog Wiki. *Getting started*. 2015. URL: http://iverilog.wikia.com/wiki/Getting_Started (cons. 20-5-2015).

[2] Nyasulu, Peter M i Knight, J. *Introduction to Verilog*. 2003. URL: http://www.cs.upc.edu/~jordicf/Teaching/secretsofhardware/VerilogIntroduction_Nyasulu.pdf (cons. 25-5-2015).

[3] 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).

[4] 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).

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