Programming with hardware/software functions

Julien Forget, Frédéric Guyomarch, Vlad Rusu

Programming with hardware/software functions

Julien Forget, Frédéric Guyomarch , Vlad Rusu

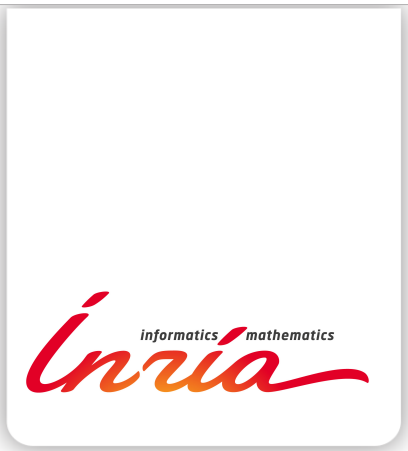
RESEARCH REPORT

N° 8835

Decembre 2015

Project-Teams Dreampal

ISSN 0249-6399 ISRN INRIA/RR–8835–FR+ENG



Programming with hardware/software functions

Julien Forget *, Frédéric Guyomarch *, Vlad Rusu†

Project-Teams Dreampal

Research Report n° 8835 — Decembre 2015 — 15 pages

Abstract: FPGAs support the implementation of a wide range of functionalities, from general-purpose processors (Softcores) to dedicated hardware accelerators (Intellectual Properties). This blurs the traditional line between software and hardware, since in many cases a functionality can be achieved either by executing code on a softcore or by running an IP. In this paper we focus on programming parallel architectures where IPs are interconnected using softcores.
We propose a programming language, called HiHope, which exploits this kind of architecture. HiHope includes constructs for switching at runtime between *hardware functions* (implemented by IPs) and *software functions* in a transparent way. It also contains constructs for executing parallel functions (either hardware or software ones) and for redefining functions dynamically.
We show how HiHope programs can be compiled for execution on master-slave parallel architectures based on the HoMade processor, a softcore processor designed as an IP integrator.

Key-words: Hardware functions, FPGA, compilation, language

* CRIStAL, University of Lille
† INRIA Lille Nord Europe

RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Programmer avec des fonctions hardware et software

Résumé : Les FPGA supportent l'implémentation d'une large game de fonctionnalités, depuis les processeurs généraux (Softcores) jusqu'aux accélérateurs matériels (IP). Cela a tendance à gommer les différences entre hardware et software, car de nombreuses fonctionnalités peuvent être réalisées ou bien en exécutant du code sur un softcore, ou bien en déclenchant un IP. Dans ce papier, nous nous concentrons sur les architectures parallèles où les IPs sont connectées en utilisant des softcores.

Nous proposons un langage de programmation appelé HiHope dédié à ce type d'architecture. HiHope inclut des constructions pour basculer dynamiquement entre des *fonctions hardware* et des *fonctions software* de façon transparente. Il contient aussi des contructions pour l'exécution parallèle de fonctions (qu'elles soient hard ou soft) et pour redéfinir ces fonctions dynamiquement.

Nous montrons comment les programmes écrits en HiHope se compilent pour une exécution sur des architectures parallèles maître-esclaves basées sur le processeur HoMade, un softcore conçu comme un intégrateur d'IP.

Mots-clés : Fonctions matérielles, FPGA, compilation, langages

1 Introduction

FPGAs (Field Programmable Gate Arrays) are a promising class of circuits for building the next generation of computers. Performance-wise, they are one or two generations behind the most advanced general purpose processors. However, they are much more flexible, since dedicated hardware can be synthesized on them for performing specific tasks, whose execution is orders of magnitudes faster than functionally-equivalent software running on a standard processor. This opens the way for new hardware architectures, consisting of softcore processors co-existing with dedicated hardware accelerators. With such an architecture, the programmer can choose between calling software functions or running dedicated Intellectual Properties (IPs) for implementing a given functionality of her program. Such choices are guided by the need to achieve speed (IPs are much faster than software code) while taking into account the intrinsic limitations of the hardware (not all functionalities of a program can be implemented in hardware due to space limitations).

We believe that new programming language concepts are required in order to efficiently exploit the potential offered by this new class of architectures. In this paper, we focus on the following concepts: hardware functions (in addition to software functions) for invoking IPs, parallel function calls (in addition to sequential function calls) and dynamic function redefinitions (in addition to static function definitions) to enable switching between hardware and software implementations at runtime. We illustrate these features in a proof-of-concept language called HiHope. We show how HiHope programs can be compiled for execution on parallel architectures based on the HoMade [6] processor and hardware IPs.

1.1 Example

In this section we illustrate HiHope on a complete example (Example 1). The program may look a bit intimidating for readers not familiar with the Forth programming style. It serves as a complete example for emphasizing the main features of the language and our descriptions focuses on those features. Simpler, more gradual examples are presented in the rest of the paper.

The program computes an approximation of $\pi/4$ based on a Monte-Carlo random method: a set of random points $M = (x, y)$ is generated, with $0 < x < 1$ and $0 < y < 1$. For each M, if $x^2 + y^2 \leq 1$ then M is part of the unit circle's first quadrant. Thus the ratio between the number of points in the quadrant (a) and the total number of points generated (b) is an approximation of $\pi/4$, i.e. $a/b \simeq \pi/4$.

The program relies on a set of auxiliary or predefined functions that or not detailed here, but summarized, along with the functions used in the other examples of the paper, in Annex A. HiHope uses a prefix notation and is stack-based: a function pops its input from the stack and pushes its outputs on the stack. For instance, at L27 `nbIt` is pushed on the stack, 2* pops this values and pushes its result ($nbIT * 2$) on the stack. The program is designed for execution on a master-slave architecture, with only two slaves for simplicity of presentation. The program is made up of one part to execute on slaves and one part to execute on the master. The master waits for the slaves to complete their start-up code, then triggers their computation (L24), collects the results (L26) and outputs them (L31). The slaves execute function `piSlave` when required by the master. This function implements the Monte-Carlo-based algorithm described previously. Let us focus on the random number generator, i.e. function `rand`. Its implementation changes *dynamically* at runtime. Initially (L5), it relies on the faster *hardware* implementation `randIP`. When a certain threshold is reached (L13), which corresponds to the maximum capacity of the hardware generator, the implementation switches to the slower software implementation `randSoft`. Notice the modularity here: the switch between the software and hardware implementations is transparent for the rest of the program.

Example 1.

```
1  slave
2    ... \ auxiliary definitions
3    : randSoft ... ;
4    ip randIP 0 1 randSoft 16
5    :dyn rand randIP ;dyn
6
7    : nbIt 0x100000 ;
8    : itIPmax 0x50000 ;
9
10   : piSlave
11       0 \ count
12       nbIt for
13         i itIPmax = if
14           dynamic rand randSoft endif
15         rand
16         split2 radius inorout
17         add
18         next
19         put
20   ;
21
22 master
23       ... \ auxiliary definitions
24   : piMaster
25       wait
26       // piSlave
27       wait
28       S2M <X S2M +
29       nbIt 2*
30   ;
31
32   piMaster
33   print print
34 endprogram
```

1.2 Related work

Several approaches that allow the mixing of software and hardware programming have been proposed. Most of these approaches are implemented in compilers, which are in charge of mapping parts of the source code to specific hardware IPs, the rest of the source being mapped to lower-level software (e.g., machine code of a processor). By contrast, in our proposal, mixed software/hardware functions are part of the programming language, so decisions to execute pieces of code in hardware or in software are taken at runtime. With respect to compile-time decisions implemented in related approaches, our runtime-based one offers more flexibility.

Related work includes GeCos [7], a C compiler that targets both machine code for Application Specific Instruction set Processors (ASIPS) and hardware accelerators on FPGAs; Molen [12], also a C compiler that targets reconfigurable FPGA-based hardware; and OrCC [13], a compiler from a data-flow language to various intermediary languages, ranging from C for executing an application in software to Verilog for hardware execution. In other environments the software is executed on software-emulated hardware. SystemC [1] allows one to execute C programs on hardware emulated at various levels of abstraction. JHDL [2] is an earlier approach, in which Java is used both as software and hardware programming language; the hardware part can be either emulated or synthesized on FPGAs. Hardware description languages such as VHDL [9] and Verilog [8] are, of course, dedicated to programming hardware. However, they share many features with software programming languages, e.g, they have been endowed with formal semantics [10, 11]. Finally, hardware-software co-design approaches (see, e.g. [5] for a survey) are more remotely related to our work; they typically consist in designing embedded systems as software together with the dedicated hardware for running it.

2 Background

This section provides a brief overview of the context of our work. We first present the target processor, HoMade. Then we give a quick reminder of the Forth language, from which HiHope is derived.

2.1 The HoMade Processor

HoMade is a softcore processor, i.e. a processor that can be synthesized on FPGAs. It is minimalist by design, so that each instance only occupies a small portion of standard FPGAs[1]. One HoMade instance can do almost nothing by itself: almost everything is delegated to IPs. The HoMade processor follows the

[1]For example, a grid network consisting of one HoMade master and 12*12 HoMade slaves fits onto a Xilinx VC707 chip, together with IPs for arithmetical/logical operations and for network communications.

`l BR`	branch to label `l`
`l BZ`	relative conditional branch to label `l` (jump if top of the stack is zero)
`l BNZ`	relative conditional branch to label `l` (jump if top of the stack is non zero)
`l CALL`	procedural call to label `l`
`RET`	return from call
`HALT`	stops the core
`x y z IP`	invoke IP `z` with `x` inputs and `y` ouputs
`l SPMD`	triggers slaves (they start at label `l`)
`WAIT`	synchronization barrier
`l i1 i2 i3 WIM`	Write instructions `i1`, `i2`, `i3` at label `l`
`NOP`	No operation
`i LIT`	Push `i` on the stack

Figure 1: The HoMade instruction set.

Harvard architecture and thus has two different memories. The first one is a standard addressed memory containing the program instructions. The second memory is a stack used for local computations and for communicating with IPs. The rest of the HoMade structure is made up of a core unit containing the instruction decoder and the control unit. There is no addressed memory for manipulating data and no ALU: these functionalities are implemented as IPs if needed. For convenience, the HoMade distribution contains a library of IPs, which includes arithmetic operations, stack manipulation, network communication, etc. However, the user is then free to use her own IPs.

The HoMade instruction set is very small and essentially contains control instructions (jumps, procedure call, etc). First, there are instructions for triggering code execution on other HoMade instances (the *slaves* of a given *master* instance). Then, there is an instruction for modifying the program memory, which makes the processor *reflexive*: it can modify the program it is executing at runtime. Last but not least, there is an instruction for triggering computation on IPs, which communicate with the processors via their stack. The full HoMade instruction set is summarized in Figure 1. Branching and function call instructions are fairly standard and are not detailed here. The remaining instructions are presented quickly below. More details can be found online [6].

Master-slave interaction The HoMade softcore was designed with the following on-board architecture in mind: one master core and several (possibly many) slave cores, with an interconnection network implemented as an IP. All communications are handled as IP invocations, performed either by a slave or by the master. The general organization and thus the network topology are decided by the user during synthesis.

Two instructions, both executed by the master, are reserved for master-slave interaction. The `SPMD` instruction orders all slaves to run a piece of program located at a common start address (referring to the memory of the slaves). The `WAIT` instruction is a synchronization barrier that detects the end of the processing of the slaves.

Dynamic rewriting of instruction memory The Write in Instruction Memory (`WIM`) instruction enables the modification of the program memory content at runtime. The instruction takes the following parameters: the memory location to be modified and the sequence of instructions to write there. In this paper, this instruction is used as the basis for implementing dynamically redefinable functions.

IP invocation Before execution, the user pre-selects a set of IPs that will be used by the application. At run-time, the IP invocation relies on a single instruction (denoted `IP`) that can trigger up to 2047 different IPs. The instruction takes three parameters: the number of input values of the IP (at most three,

popped from the stack), the number of output values (at most three, pushed on the stack) and finally the identification number of the triggered IP. If the IP number is less than 1024, it indicates that the IP runs within a single clock cycle. If the number is more than 1024, it means that the IP takes more than one cycle. In this case, the IP raises a signal at the end of its execution.

2.2 Forth

HiHope is strongly inspired from Forth [4], a procedural imperative language that has several interesting features for programming on HoMade. This includes *reflection* (ability to modify the program structure during program execution), *extensibility* (the programmer can create new instructions at runtime) and also being *stack-based*. Before presenting HiHope, we provide a quick presentation of Forth and its terminology in this section.

A subroutine in Forth is called a *word* and Forth programs consist of a succession of word *definitions* and word *calls*. Let us consider the following simple example evaluated by a Forth interpreter:

```
1  : POSITIVE DUP 0 > IF 1 ELSE 0 THEN ;
2  ok
3  5 POSITIVE .
4  1 ok
5  : POSITIVE DUP 0 > IF ELSE DROP THEN ;
6  ok
7  3 POSITIVE .
8  3 ok
```

Line 1 defines the word `POSITIVE` (definition starts with "`:`" and ends with "`;`"). This word first duplicates (`DUP`) the value on top of the stack, let us say x, then pushes 0, compares x and 0 (poping both in the process), and based on the comparison pushes either 1 or 0 on the stack (`IF` also pops the comparison value). In Line 3, we push 5, call `POSITIVE`, and pop and print the top of the stack with ".". Line 4 shows the result of the evaluation of Line 3. On Line 5, we redefine `POSITIVE` so that it does nothing if the stack top is positive and drops it otherwise. This overwrites the previous definition of `POSITIVE` as shown by the interpreter result at Lines 7-8.

3 The HiHope Language

HiHope is a very small language that follows the Forth syntax and structure closely, to the point that it can easily be emulated in Forth[2] (which can be useful for early prototyping). It keeps a subset of Forth words (loops, conditionals, word definitions, ...) and introduces new words dedicated to IP invocation, dynamic redefinition and parallel execution. HiHope has an intermediate level of abstraction: higher than machine code, lower than modern general purpose programming languages. Being close to the underlying hardware platform, it can serve as a HoMade assembly language for higher level languages, in the same way as C code is produced by Synchronous languages compilers for instance [3].

As emphasized previously, Forth exhibits interesting features for programming HoMade and thus was a natural inspiration for HiHope. However, we chose to not directly program HoMade with Forth for two reasons. First, there is obviously no Forth compiler for HoMade and we only need a reduced language to program HoMade, while the ANSI Forth standard contains more than 300 words. Second, there is no standard support for parallel execution and IP invocation in Forth.

3.1 Main concepts

In this section we introduce HiHope through a set of simple examples. The examples have been designed for a Nexys 3 board (by Digilent ®) but can easily be applied to any board containing the following

[2] We provide an emulator at `blah`

components (or equivalent ones):

- A FPGA, to embed the HoMade architecture;
- A set of switches that serves as the input device. The `switch` IP reads a value defined by the position of the switches and outputs this value on the stack;
- A seven-segment or LCD display that serves as the output device. The IP `print` prints the top of the stack on the display.

Hardware functions An IP (or hardware function) is defined by its name, the number of inputs/outputs, an emulation function and a unique integer identifier. Once defined, it can be invoked simply by writing its name. This is illustrated below:

Example 2.

```
1  ip switch 0 1 dummy 4
2  ip print 1 0 dummy 2
3  ip dup 1 2 dummy 8
4  ip swap 2 2 dummy 9
5  ip 0< 1 1 dummy 51
6  ip - 2 1 dummy 33
7
8  : abs dup 0< if 0 swap - endif ;
9
10 master
11   switch
12   abs
13   print
14 endprogram
```

In this example, we first declare several IPs. The `switch` IP has no input and one output, its identifier is 4 and we ignore the emulation function here (`dummy`). Similarly, we declare the IPs `print`, presented previously, `dup`, which duplicates the top of the stack, `swap`, which swaps the two top values of the stack and `0<`, which tests if a value is negative. Then, we define a software function that computes the absolute value of an integer (`: abs ... ;`). The program is executed on a single master core (`master ... endprogram`) and consists in reading a value, computing its absolute value and outputting the result. Notice that hardware and software functions are invoked exactly with the same syntax (though they will we compiled in different ways).

Dynamic function redefinitions A specific syntax is used to declare functions that can be redefined at run-time (`:dyn ... ;dyn`). Using the word `dynamic`, the programmer can then rebind the name defined that way to a new function (either software or hardware), at any point of the master or slave program:

Example 3.

```
1  ip print 1 0 dummy 2
2  ip switch 0 1 dummy 4
3  ip 0= 1 1 dummy 50
4  ip + 2 1 dummy 32
5  ip - 2 1 dummy 33
6
7  :dyn arit - ;dyn
8
9  master
10
11   switch
12   0=
13   if
14     dynamic arit +
15   endif
16   3 4 arit
17   print
18 endprogram
```

In this example, the behaviour of the `arit` function changes during execution. Its initial behaviour is to perform a subtraction. The program then reads a value and if it equals 0 `arit` becomes an addition. Here, the behaviour of `arit` switches between two hardware functions (IPs). We could also switch from a hardware function to a software function (as it was the case in Example 1) or from a software function to another software function, using exactly the same syntax.

Parallel execution HiHope follows a master-slave model of execution. The master can trigger the execution of a function on the slaves using word `//` and wait for the slaves to complete their current computation using word `wait`. It can also continue its computations while slaves do their own. The programmer can specify some code that will be executed at slave start-up:

Example 4.

```
1  ip put 1 0 dummy 0x201
2  ip S2M 0 1 dummy 0x012
3  ip xnum 0 1 dummy 0x402
4  ip ynum 0 1 dummy 0x403
5  ip <X 0 0 dummy 0x1FA
6
7  ip + 2 1 dummy 32
8  ip print 1 0 dummy 2
9
10 slave
11   : putDiag
12       + put
13   ;
14
15   xnum ynum
16
17 master
18   : print2
19       S2M
20       print
21       <X
22       S2M
23       print
24   ;
25
26   wait
27   // putDiag
28   wait
29   print2
30
31 endprogram
```

This example assumes a topology consisting of a master and a torus grid of slaves. At start-up, every slave pushes its x and y coordinates in the grid on their own stack (`xnum`, `ynum`). The master first waits for the initial slave operations to complete, then triggers the execution of function `putDiag` on the slaves. This causes slaves to sum their x and y coordinates and output the result on the communication network (using IP `put`). The master then waits for the slaves and outputs the values produced by the first two slaves using function `print2`. The `S2M` (slave-to-master) IP transfers data produced by the first slave on the communication network to the master stack. `<X` shifts all values in the network according to the x axis, thus the subsequent `S2M` transfers the value of the second slave to the master stack. Notice how network operations are performed by dedicated IPs. Also notice that the language allows to apply "`//`" directly on an IP (e.g. `// +`).

3.2 Syntax

The grammar of HiHope is given below. In order to simplify the presentation of the compilation, we assume that words are defined locally (either on the slaves or on the master) and not globally, even though the complete language supports both.

< *hiprog* >	::=	**slave** *definition* *word* **master** *definition* *word* **endprogram**
< *definition* >	::=	*ip_def* | *static_def* | *dynamic_def* | *definition definition*
< *ip_def* >	::=	**ip** *name int int name int*
< *static_def* >	::=	: *name* [**recursive**] *word*;
< *dynamic_def* >	::=	: **dyn** *name* [**recursive**] *word* ;**dyn**
< *word* >	::=	*literal* | *condition* | *loop* | **wait** | **nop** | *dyn_redef* | *call* | // *call* | *word word*
< *call* >	::=	*name*
< *dyn_redef* >	::=	**dynamic** *name call*

A HiHope program is made up of one part to be executed on the **slave** processors and one part to be executed on the **master** processor. Both parts have the same structure: a list of auxiliary definitions

followed by a list of words. The word list represents the code to execute. A *definition* can either be an IP, a static (software) definition that cannot be redefined dynamically, or a dynamic definition. An IP (*ip_def*) is defined by its name, its number of inputs/outputs (given as an integer value), an emulation function (the last *name* field) and the IP unique identifier (an integer). The syntax of a *static definition* is exactly that of a Forth word definition. *Dynamic definitions* are introduced by distinct keywords because they are not compiled the same way as static definitions.

Words consist of the usual literals, condition and loop structures, the **wait** that enables the master to wait until slaves complete their latest parallel call, a **nop** (no operation) and finally, dynamic redefinition, word call and parallel call. A *call* is syntactically just a name, the compiler will choose based on previous definitions whether it must be compiled as an IP call or as a regular software word call. Similarly, a *parallel call* (`// call`) can be applied to either an IP or a regular definition. A *dynamic redefinition* `dynamic def-name call-name` assigns `call-name` as the new behaviour for `def-name`. Again, notice that `def-name` can correspond to either an IP or a regular definition.

4 Compilation

The compiler front-end is quite small since it does not contain complex static analyses. We mainly focus on the compiler back-end, which operates in two steps. First, the HiHope program is translated into a list of labeled HoMade instructions. Then, this list is translated into actual machine code. This translation consists in replacing labels by physical addresses and producing the binary code corresponding to the instructions. The translation scheme described in this section has been implemented in the JHoMade toolset[3].

4.1 Front-end

The compiler front-end is composed of three steps: syntax analyses, checking for undefined words and building a *Definitions Environment*. The first two steps are pretty straightforward and thus are not detailed here. The Definitions Environment maps definition names to 1) their kind (either *hard* or *soft*) 2) the context in which they are called, (either a *parallel* call, a *local* call, or *both*). This information is used during the translation of word calls. It is built in a single code traversal. The kind of a definition is simply determined by the syntax of its declaration. The call context is assigned as follows:

- *parallel* if the definition name appears in a parallel call on the master;
- *local* if the definition name appears in a local call (either on the master or slave);
- *both* if both conditions are true (which is possible only on the master).

4.2 From HiHope to HoMade

In this section we detail the translation of each HiHope construct into HoMade instructions. The translation is equipped with a label generator (omitted here for the sake of conciseness), which generates fresh labels for the program instructions. Labels generated that way are denoted L_i. Slave labels and master labels are in different name-spaces thus duplication is allowed. We also define some auxiliary functions:

- $end(local) = \texttt{RET}$, $end(parallel) = \texttt{HALT}$;
- $label(id, local) = id$, $label(id, parallel) = id\_par$;
- $ip(ipName)$, which returns the IP call corresponding to the IP $ipName$. For instance, ip(print)=`1 0 short 2 IP`, since `print` has 1 input, 0 outputs, is a short IP and has number 2.

[3] Available freely at `https://gforge.inria.fr/frs/?group_id=3646&release_id=8294`.

Invocation	**Result**
`a dataPush`	Push `a` on the data-stack.
`a dataPushDup`	`a`, push `a` on the data-stack leave *a* on HoMade stack.
`dataPop`	`a`, pop `a` from data-stack to HoMade stack.
`dataPopDrop`	Pop from data-stack, nothing to HoMade stack.

Figure 2: The data-stack IP set.

HiHope	HoMade
literal	L_1 : *literal* LIT
WAIT	L_1 : WAIT
NOP	L_1 : NOP
IF W_1 ELSE W_2 ENDIF	L_1 : *ip*(drop) L_2 : L_5 BZ *translate*(W_1) L_3 : L_5 BR L_4 : NOP *translate*(W_2) L_5 : NOP
I	L_1 : *ip*(dataPop) L_2 : *ip*(dataPushDup)
J	L_1 : *ip*(dataPop) L_2 : *ip*(dataPop) L_3 : *ip*(dataPushDup) L_4 : *ip*(swap) L_4 : *ip*(dataPush)

HiHope	HoMade
BEGIN W UNTIL	L_1 : NOP *translate*(W) L_2 : *ip*(drop) L_3 : L_1 BZ
FOR W NEXT	L_1 : *ip*(dataPush) L_2 : NOP *translate*(W) L_3 : 0 L_4 : *ip*(dataPop) L_5 : *ip*(1-) L_6 : *ip*(dataPushDup) L_7 : *ip*(>) L_8 : *ip*(drop) L_9 : L_2 BZ L_{10} : *ip*(popDrop)

Figure 3: Translation of basic constructs.

Basic constructs (Figure 3) HiHope contains several variants of loops and conditionals, which are not all presented here. For the most part, their translation rules are fairly straightforward. However, following the HoMade philosophy, the translation patterns are based on some IPs. Consider the `for... next` construct for instance. As for the compilation of counted loops in Forth, our translation uses an auxiliary data-stack (not to be confused with the HoMade stack) to store the loop indices. This data-stack is implemented through a set of IPs, summarized in Figure 2. In more details:

- First, note that "`n for ... next`" iterates $n+1$ times and that the loop index produces values starting with n and ending with 0. The loop bound is stored on the data-stack before starting the loop (L_1), decremented and tested at the end of the loop ($L_4 - L_7$) and cleaned-up after the loop (L_{10});
- `I` is actually a HiHope keyword. Its translation consists in copying the top of the data-stack to the HoMade stack;
- The translation of `J` is similar but slightly more complex since it is the second value on the data-stack.

Notice that conditional branching instructions (`BZ` and `BNZ`) are preceded by a `DROP` instruction. This is an optimization due to the instruction pipeline of HoMade: the branching condition is evaluated based on the value that was on top of the stack two cycles earlier, thus we can add an instruction (here `DROP`) between the production of the condition value and the branching instruction.

Program (Figure 4) The same rule applies for translating the code of the slave and the code of the master (which are generated as two separate parts). Definitions are translated first, then words. The program branches over the code produced for the definitions and halts at the end of the execution of the words.

HiHope	HoMade
$D_1\ D_2$	$translate(D_1)$ $translate(D_2)$
$W_1\ W_2$	$translate(W_1)$ $translate(W_2)$
slave $D\ W$ (or) master $D\ W$ endprogram	entryL : startP BR $translate(D)$ startP : NOP $translate(W)$ end : HALT

Figure 4: Translation of a complete program.

Hardware/software functions (Figure 5) The HoMade code produced for Example 2, presented previously, is given below:

Example 5.

```
  entryL        : startP BR
  startP        : NOP
  end           : HALT
#master
  entryL        : startP BR
  abs           : NOP
  L6            : 1 2 short 8 IP
  L7            : 1 1 short 33 IP
  L10           : 1 0 short 0 IP
  startIf8      : endIf9 BZ
  L11           : 0 LIT
  L12           : 2 2 short 9 IP
  L13           : 2 1 short 21 IP
  endIf9        : NOP
  abs_end       : RET
  startP        : NOP
  L15           : 0 1 short 4 IP
  L16           : abs CALL
  L18           : 1 0 short 2 IP
  end           : HALT
```

The code from `abs` to `abs_end` results of the translation of the software function `abs`. The subsequent code is generated for the list of words executed by the master. There are several instances of code generated for a call to a hardware function (e.g. at `L6`) or to a software function (at `L16`). In more details, the translation rules for hardware/software functions work as follows:

- An IP declaration produces no code, it is just stored along with its parameters in the Definitions Environment;
- The translation of a static definition varies slightly depending on the context in which it is called in the program. A definition ends with a return if it is called locally, while a definition called in parallel, i.e. triggered on a slave by the master, halts until some other computation is triggered by the master. If the definition is called in both contexts, it is translated twice (once with $ctxt = local$ and once with $ctxt = parallel$);
- The compiler translates a call based on its kind, stored in the Definitions Environment. A call to a software function is translated into a call to the starting label of the code generated for the corresponding definition. A call to a hardware function (IP) is directly translated into a HoMade IP invocation.

Dynamic redefinition (Figure 5) The code obtained for Example 3 is the following:

Example 6.

```
entryL        : startP BR
startP        : NOP
end           : HALT
#master
entryL        : startP BR
arit          : arit_body CALL
arit_ret      : RET
arit_body     : NOP
L9            : 2 1 short 21 IP
arit_retbody: RET
startP        : NOP
L10           : 0 1 short 4 IP
L11           : 1 1 short 32 IP
L14           : 1 0 short 0 IP
startIf12     : endIf13 BZ
L15           : arit 2 1 short 20 IP
                  RET NULL  WIM
endIf13       : NOP
L17           : 3 LIT
L18           : 4 LIT
L19           : arit CALL
L21           : 1 0 short 2 IP
end           : HALT
```

The instructions from `arit` to `arit_retbody` result of the translation of the dynamic definition `arit`. The code for the dynamic redefinition is located at `L15`. In more details, the translation rules for dynamic redefinitions work as follows:

- Similarly to static software definitions, the translation of a dynamic definition varies slightly depending on its call context (return/halt). The body of the dynamic definition is executed using a `CALL`, thus there is a double indirection when calling a dynamic definition. The reason for this lies in the translation rule for dynamic redefinitions;
- The code produced for `dynamic arit +` will rewrite at run-time the code located at the label of `arit`, replacing the code at this location by a call to `+`. This is the reason for the double indirection produced for a dynamic definition: a call to `arit` will always jump to the same location (the label of `arit`). However, the call located there can change, depending on the dynamic redefinitions that have been executed thus far. The generated code differs slightly depending on whether the new behaviour corresponds to a software definition or to an IP because the call is not performed by the same instruction and also because the parameter of a WIM must always be exactly three words long.

Parallel execution (Figure 5) The code obtained for Example 4 is the following:

Example 7.

```
entryL          : startP BR
putDiag_par     : NOP
L3              : 2 1 short 20 IP
L4              : 1 0 short 201 IP
putDiag_par_end: HALT
startP          : NOP
L6              : 0 1 long 402 IP
L7              : 0 1 long 403 IP
end             : HALT
#master
entryL          : startP BR
print2          : NOP
L12             : 0 1 short 12 IP
L13             : 1 0 short 2 IP
L14             : 0 0 short 1fa IP
L15             : 0 1 short 12 IP
L16             : 1 0 short 2 IP
print2_end      : RET
startP          : NOP
L18             : WAIT
L20             : putDiag_par SPMD
L21             : WAIT
L22             : print2 CALL
end             : HALT
```

On the slave, the code from `putDiag_par` to `putDiag_par_end` results of the translation of function `putDiag`. Since this function is triggered by the master, it ends with a `HALT` that signals completion to the master. The code on the slave from `startP` to `end` corresponds to the code executed at start-up, before the master starts triggering other computations. The code corresponding to the parallel call on the master is located at label `L20`. In more details, the translation rules for parallel calls work as follows:

- A parallel call to a software function is translated into a `SPMD` instruction. Notice that the label invoked by the `SPMD` corresponds to a label located on the slave, not on the master;

HiHope	Environment	HoMade
ip id in out emu hex	$E \cup \{id \mapsto (hard(in,out,hex),...)\}$	
: id [recursive] W ;	$E \cup \{id \mapsto (soft,ctxt)\}$	$label(id,ctxt)$: NOP $translate(W)$ L_1 : $end(ctxt)$
ID	$E \cup \{id \mapsto (soft,...)\}$	L_1 : $label(ID,local)$ CALL
ID	$E \cup \{id \mapsto (hard(in,out,hex),...)\}$	L_1 : in out hex IP
// ID	$E \cup \{id \mapsto (soft,...)\}$	L_1 : $label(ID,parallel)$ SPMD
// ID	$E \cup \{id \mapsto (hard(in,out,hex),...)\}$	(Code on slave) ID_parcall : NOP L_1 : in out hex IP L_2 : HALT (Code on master) L_3 : ID_parcall SPMD
:dyn id [recursive] W ;dyn	$E \cup \{id \mapsto (soft,ctxt)\}$	$label(id,ctxt)$: id_body CALL L_1 : $end(ctxt)$ id_body : NOP $translate(W)$ L_2 : RET
dynamic dynID ID	$E \cup \{id \mapsto (soft,ctxt)\}$	L_1:$label(dynID,ctxt)$ ID CALL WIM
dynamic dynID ID	$E \cup \{id \mapsto (hard(in,out,hex),ctxt)\}$	L_1 : $label(dynID,ctxt)$ in out hex IP $end(ctxt)$ NULL WIM

Figure 5: Translation of constructs related to function calls.

- In the case of a hardware function being called in parallel, since the `SPMD` parameter must correspond to a label on the slave, the compiler adds code on the slave just as if the IP call was packed inside a static definition.

4.3 Producing machine code

The code obtained by the translation rules of the previous section is translated into machine code in 3 steps. First, the compiler performs simple optimizations: removing useless `NOP` instructions, which are introduced by some translation rules for simplification, and inlining calls that branch to a sequence with at the most three instructions (also applied to the calls generated for dynamic redefinitions to avoid some double call indirections). Second, labels are translated to machine addresses. Third, instructions are translated to binary code in a one-to-one translation, which basically replaces each instruction by the corresponding binary code.

5 Conclusion

We presented the HiHope language and its compilation. HiHope is designed for programming parallel hardware architectures, consisting of IPs interconnected using HoMade softcores, and implemented on FPGAs. At the heart of the language lies the objective of transparently mixing traditional software functions with hardware functions (implemented as IPs).

Currently, HiHope enables to change the implementation of a given function at run-time, replacing it by a different software or hardware implementation. We are working to enable the dynamic reconfiguration of IPs on the board, so that the FPGA area of unused IPs (unused at some point of the execution) can be used to load a different set of IP. This will enable to alter the IP set dynamically depending on the dynamic requirements of the application.

References

[1] Accelera Software. System C standard. available at `http://www.accellera.org/downloads/standards/systemc`.

[2] P. Bellows and B. Hutchings. JHDL-an HDL for reconfigurable systems. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 175–184. IEEE, 1998.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. 91(1), 2003.

[4] L. Brodie. *Starting FORTH*. Forth, 1980.

[5] G. De Micheli, R. Ernst, and W. Wolf. *Readings in hardware/software co-design*. Morgan Kaufmann, 2002.

[6] J.-L. Dekeyser. HoMade. available at `https://sites.google.com/site/homadeguide/home`.

[7] A. Floch, T. Yuki, A. El Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, et al. Gecos: A framework for prototyping custom hardware design flows. In *SCAM*, pages 100–105, 2013.

[8] IEEE. Standard Verilog language reference manual. available at `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7578`.

[9] IEEE. Standard VHDL language reference manual. available at `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4772738`.

[10] C. D. Kloos. *Formal Semantics for VHDL*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[11] P. O. Meredith, M. Katelman, J. Meseguer, and G. Roşu. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*, pages 179–188. IEEE, 2010.

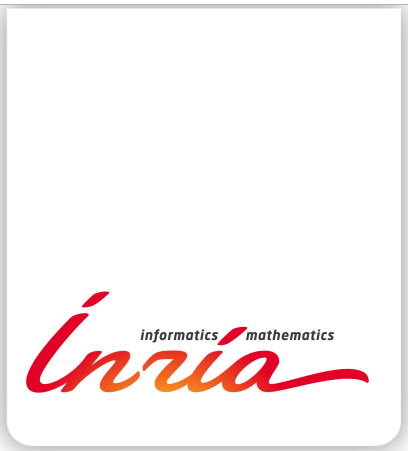
[12] E. M. Panainte, K. Bertels, and S. Vassiliadis. The Molen compiler for reconfigurable processors. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(1):6, 2007.

[13] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet. Orcc: Multimedia development made easy. In *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pages 863–866. ACM, 2013.

A Auxilliary functions

This annex lists the functions (hardware or software) used throughout the paper and provides a quick description for each.

Invocation	Result
`a dup`	`a a`
`a drop`	
`a b c rot`	`b c a`
`a b swap`	`b a`
`a 2*`	`2*a`
`a 1-`	`a-1`
`a 0<`	`a < 0`
`a print`	Display `a` on 7-segment/LCD
`switch`	`a`, read `a` from switches
`a M2S`	Transfer `a` from master to network (slave 0)
`S2M`	`a`, transfer `a` from network (slave 0) to master
`a put`	Transfer `a` from slave to network
`get`	`a`, transfer `a` from network to slave
`>X`	Shift values to the right in network
`<X`	Shift values to the left in network
`randSoft`	`a`, random value (software version)
`randIP`	`a`, random value (hardware version)
`a b radius`	`r`, distance from (0,0) to (a,b)
`d inorout`	1 if `d` is a distance in unit circle's first quadrant 0 otherwise
`a split2`	`b c`, `b` is msb and `c` is lsb^4 of `a`



RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399