Design of highly parallel architectures with Alpha and Handel

Florent Dupont de Dinechin, Tanguy Risset
INRIA/LIP ENS-Lyon
46 allée d'Italie
69364 Lyon Cedex 07, France

M. Manjunathaiah, Mike Spivey
OUCL, Parks Road
Oxford, OX1 3QD, UK

July 1, 2002

Abstract

We propose a bridge between two important parallel programming approaches in the context of reconfigurable computing: the data parallel approach and communicating sequential processes approach. Using the data parallel approach we generate a pipelined design by transforming a high level specification in the MMAlpha design framework. This design is then rapidly turned into an FPGA implementation by applying a hardware compilation framework derived from the CSP formalism. The design and implementation of such highly parallel architectures is obtained by an automatic translation process derived from a bridge between the semantics of the languages Alpha and Handel.

1 Introduction

Increasingly high level approaches are being adopted in the design/codesign of high-performance embedded systems due to the complexity in implementing complete systems in silicon or as a mixture of hardware and software. Many applications are both control and compute intensive in nature and hence highly parallel solutions are sought to meet real-time constraints. Constructing a design which maximizes the available parallelism is a difficult task. Additionally, designers are faced with the problem of designing complex systems more rapidly. The natural solution to this problem is to use automatic or semi-automatic compilation techniques from high level specifications. One of the main difficulties in this process is the definition of a correct computation model that can express high level functional specifications as well as final target designs and implementations. Existing tools use a particular computation model that can express parallelism (because hardware is targeted for performance) for a specific granularity [2]. Among the existing models, one can isolate two important models: the data parallel model and the communicating sequential processes model.

The data parallel model led to the definition of HPF and the development of methodologies for the design of highly parallel or pipelined architecture [8]. The applications targeted, which are usually compute intensive, are composed of regular and repetitive operations on large sets of data (signal processing, multimedia, etc.). Research at Irisa (Rennes) produced the MMAlpha system [3] based on the Alpha language [1] which is dedicated to the hardware synthesis of regular (systolic-like) architectures from high level (functional) specifications. A fundamental feature of this methodology is that the parallelism is implicit and the design process provides transformations to introduce concurrency in a controlled and (semi)- automatic way.

The communicating sequential processes model [4] lead to two important languages: Occam [6] for concurrent software design [6] and Handel for concurent hardware design [7]. The Handel family of languages

(HandelC [7], HandelB [11]) is particularly targeted at implementing programs in hardware. The Handel formalism is well suited for expressing complex control structures; any concurrent program can be targeted, and the hardware compilation scheme generates an equivalent hardware circuit which can be implemented on an FPGA. However, in contrast with the data parallel model the concurrency is explicit and the designer has to specify the concurrency using the language features. This often makes it difficult to design highly parallel systems as the complexity of the design increases a lot with the amount of parallelism and communication. Therefore for a class of concurrent designs, such as pipelined architectures, it would be interesting to combine the two design methodologies.

In this paper, we propose a way of easily and rapidly generating correct highly parallel designs by bridging the two design formalisms. The idea is to use the systolic synthesis methodology, which has been implemented in the MMAlpha tool, to provide correct specifications of pipelined architectures and to use the hardware compilation methodology provided by the Handel compilers to automatically translate them into hardware circuits. As these architectures are expressed in the Alpha language which has a very precise semantics, we are able to define a translation into Handel which can be guaranteed to be correct, provided that the design has some well defined characteristics. This integrated design methodology will improve the design time of implementing pipelined architectures on FPGAs.

2 Hardware compilation paradigms

In this section we highlight the differences between concepts used in Alpha and Handel. We show that the design methodologies and the resulting hardware of these two tools are strongly influenced by their underlying computation models. We illustrate this by the Finite Impulse Response (FIR) filter which is a simple convolution filter used in many digital signal processing algorithms (correlation, adaptive filtering, etc.). The problem is the following: given a possibly infinite sequence of values $x(n)$, $(n \geq 0)$ and N weights $w(i)$, $(0 \leq i \leq N-1)$, compute the sequence $y(n)$, $(n \geq N-1)$ where

$$y(n) = \sum_{i=0}^{N-1} x(n-i)w(i) \quad . \tag{1}$$

2.1 Data parallel formalism

The Alpha specification of the FIR filter is shown in Figure 1. M is the number of inputs $x(n)$, and all variables are represeted by 16-bit integers (as we will see later, we can also express the algorithm at the bit level). This specification is obtained very easily from the equation (1), it has no specified order of execution.

The MMAlpha system [3] can automatically generate a systolic architecture expressed in VHDL from this specification. The resulting architecture is sketched in figure 2. The architecture can be completely described (both its structure and its behaviour) *in the Alpha language*. For instance, one intermediate form of the program is the *space-time representation* of the program shown in Figure 3 . In this representation, index t can be interpreted as time and index p can be interpreted as a processor number. It means that the first index of each variable represents a counter on a global clock regulating the array. This assumption leads to a natural hardware target: a globally synchronous digital circuit in which all the registers are controlled by a common virtual clock. The virtual clock can be implemented using the clock enable signal on Xilinx FPGA for instance. In practice, it usually corresponds to the actual clock regulating the FPGA but can be used to freeze the execution if, for instance, the input data is not ready. Hence this virtual clock can be seen as a *global clock enable*. This strong assumption allows the designer to implement regular designs

```
system fir : {N,M | 3<=N<=M-1}
   (x : {n | 1<=n<=M} of integer[S,16];
    w : {i | 0<=i<=N-1} of integer[S,16])
      returns (res : {n | N<=n<=M} of integer[S,16]);
var
    Y : {n,i | N<=n<=M; -1<=i<=N-1} of integer[S,16];
let
    Y[n,i] = case
                { | i=-1} : 0[];
                { | 0<=i} : Y[n,i-1] + w[i] * x[n-i];
                esac;
    res[n] = Y[n,N-1];
tel;
```

Figure 1: Specification of the FIR filter in Alpha. This specification is directly obtained by serializing the summation in equation (1).

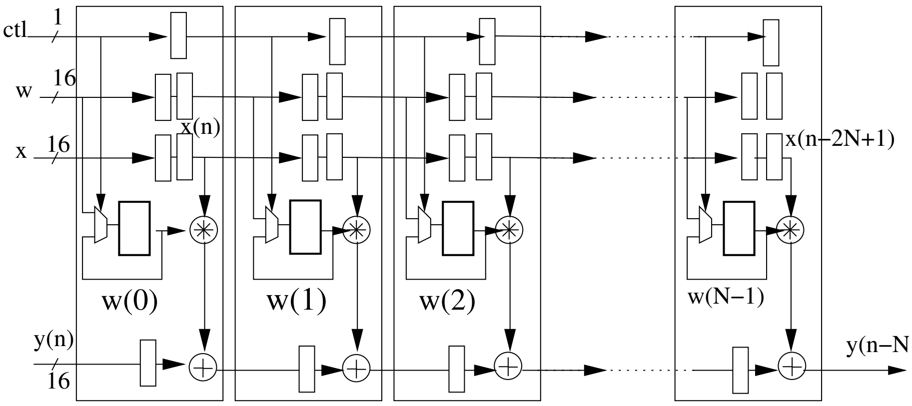

Figure 2: RTL description semi-automatically derived from this specification of figure 1 with the systolic design methodology implemented in the MMAlpha environment.

very efficiently, with minimal control overhead (control signal are usually pipelined in the array). Moreover, this control mechanism can be generated automatically because of restrictions on input format (uniform program) and hence the design can be formally proven.

Another important characteristic of the Alpha language is that all variables represent *signals* and any piece of hardware is represented by an operation between signals: operators (for combinatorial logic) or dependencies (for registers or simple connexion between signals). The architecture compiled with MMAlpha into the Alpha0 format which is a restriction of the Alpha langage on which some interpretation has been added. This compilation process is detailed in [3].

```
 system fir :   {N,M | 3<=N<=M-1}
               (x : {n | 1<=n<=M} of integer;
                w : {i | 0<=i<=N-1} of integer)
       returns (res : {n | N<=n<=M} of integer);
var
  xPipe : {t,p | 2p+1<=t<=p+M; 0<=p<=N-1} of integer;
  wPipe : {t,p | p+N<=t<=p+M; 0<=p<=N-1} of integer;
  Y : {t,p | p+N+1<=t<=p+M+1; -1<=p<=N-1} of integer;
let
  xPipe[t,p] =
      case
        { | p=0} : x[t-p];
        { | 1<=p} : xPipe[t-2,p-1];
      esac;
  wPipe[t,p] =
      case
        { | t=p+N} : w[t-N];
        { | p+N+1<=t} : wPipe[t-1,p];
      esac;
  Y[t,p] =
      case
        { | p=-1} : 0[];
        { | 0<=p} : Y[t-1,p-1] + wPipe[t-1,p] * xPipe[t-1,p];
      esac;
  res[n] = Y[n+N,N-1];
tel;
```

Figure 3: Body of the Alpha program of figure 1 after scheduling and mapping in the MMAlpha environment (i.e. space-time representation).

2.2 Communicating sequential processes

A possible Handel specification of the FIR of equation (1) is shown in Figure 4 (we used the HandelC formalism here). This specification is very close to a C formulation but it does not contain any parallelism[1]. Unlike the Alpha language, the semantics of the Handel program does not change; Handel is not a transformation environment, parallelism has to be expressed explicitly. The hardware compiled from the program of Figure 4 with the Handel compiler will respect some important guidelines [12, 7]. The compiler generates registers as resources for (static) arrays and a RAM module (memory and addressing logic) for dynamic arrays. A key feature of the language is that each assignment will take exactly one clock cycle, *provided it can be executed.* In practice, it means that each variable will be stored in a register which is itself controlled by a boolean signal. Hence, unlike in Alpha, all variables represent registers in the Handel formalism (but Handel also permits to express signals with `always` construct in HandelB and the `signal` type in HandelC). For instance, on Figure 4, statement `y[n][0] = 0` is scheduled on, say clock cycle 0, then `i=1` is schedule on clock cycle 1, the assignement to `y[n][1]` on clock cycle 2, then `i++` on clock cycle 3 and so on.

A global clock is continuously applied to all registers (synchronous design) but a *control token* mecha-

[1]Of course one could come up with an equivalent parallel program, but that is what we aim to achieve using the data parallel approach and we demonstrate it in a later section.

nism is added to express the flow of control (control states) dynamically at run-time. The control tokens are used to schedule the assignment and communication statements in the language [7]. This control token acts like a *local clock enable*.

To illustrate the difference in the two models, we have shown on Figure 5 a simple piece of hardware as it would be implemented from the MMAlpha system (a) and from the Handel compiler (b). A 5 bit signal `r` gets the value of one of the two registers: `a` or `b`. Note that in the implementation from MMAlpha, there is a simple global clock enable, but the control of the multiplexer is done explicitly while in the handel mechanism, the clock enable may be different for each signal and the control of the multiplexer is done using these clock enable signals. Note also that, if the MMAlpha hardware is slightly more difficult to generate (because it implies the correct setting of the `ctrl` signal), the resulting complexity is lower than the Handel implementation complexity (because the clock enable mechanism is simpler.

Equivalence between Handel hardware and MMAlpha hardware will be very difficult to prove if the Handel program uses channels, hence we will not use channels in the architecture expressed in Handel (we will use then to bring the input data to the architecture). We would like to highlight the following: although the implementation are different, the two designs are compatible. It means that they will have the same behaviour (provided the control signal `ctrl` is correctly set up in the MMAlpha hardware generation).

```
void main (void) {
  int 16 x[6];
  int 16 w[3];
  int 16 Y[4,4];
  int 16 res[4];

    for (n = 0; n <= 5; ++n) {
        Y[n][0]=0;
        for (i = 1; i <= 2; ++i) {
            Y[n][i]=Y[n][i-1]+w[i]*x[n-i]
        }
    }
}
```

Figure 4: A possible Handel program for the FIR filter of equation (1).

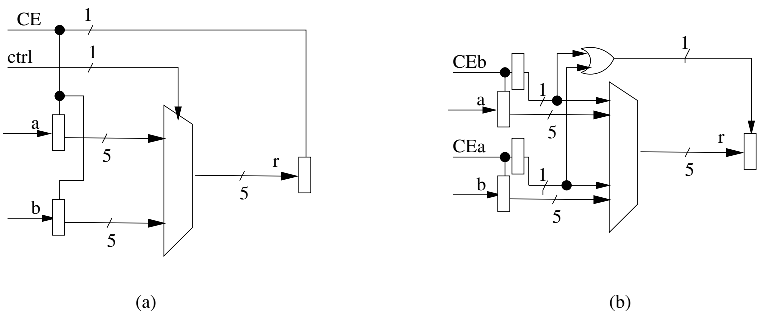

Figure 5: Different hardware generated from MMAlpha (a) and Handel (b) having the same behaviour (provided `ctrl` is correctly set up)

3 Combining the models

We have highlighted in the previous section the main differences in the underlying semantics of Alpha and Handel: global clock enable versus local clock enable control and variables implemented as signal versus register (we do not consider channels here). If we want to translate an architecture expressed in Alpha0 into Handel we have to solve to two following problems: translate registers (which are operators in Alpha) into variables and minimize the control overhead that will be introduced by the local clock enable Handel mechanism. We will explain that in section 3.2.

In the following section, we describe how to bridge the Alpha and Handel languages in deriving pipelined designs. As it involves several tools in the design process, we refer to this as *co-design*, although it may not be hardware/software co-design but rather hardware/hardware co-design. As in a classical co-design framework, we first need a co-simulation step in which the functionality of the design is verified and we then need a co-synthesis step. These two steps correspond to two translations from Alpha to Handel. In the co-simulation step we use the first stage of the translation from the scheduled Alpha program and in the co-synthesis step we use a more refined architecture with detailed control circuits derived from the Alpha0 translation. In both stages the whole system can be simulated or synthesized with Handel. The difference between the two steps will correspond to two different stages in the above mentioned design steps of MMAlpha (section 2.1)

3.1 Co-simulation

The first architectural description is obtained after the *space-time* transformation. Co-simulation can be applied as soon as the Alpha program has been scheduled and we have this initial architecture. Scheduling attempts to find an execution date for each computation (the program shown in Figure 3 is scheduled). Here, each Alpha equation takes one clock cycle hence the scheduled program can be easily translated into Handel, the main difficulty being to translate polyhedral domains into loops. This non-trivial problem is already solved in the Alpha to C compiler [10, 9]). For this example, the resulting Handel program will contain an output sequential replicator (loop on index t) and many inner parallel replicators, (loops on index i). This structure of the resulting Handel program is shown in Figure 6.

We refer to this translation as "naive" because the efficiency of the resulting Handel program is not addressed — only a semantically equivalent program is derived. Indeed, this translation does not attempt to reuse memory. As Alpha is a single assignment language, the amount of memory used in the resulting Handel program can be quite large. However, it has the same functional behavior as the final architecture. Functional validity can be verified by simulating the translated program using different simulators (source level and circuit level simulators) and checking it against the execution of the C code output from the MMAlpha code generator.

3.2 Co-synthesis

After the functional validation of the design in both MMAlpha and Handel has been established, we look for a more efficient implementation of the Alpha architecture in the Handel formalism. More specifically, we aim to translate a more detailed design from the Alpha0 format in which control signals are made explicit. Figure 7 shows the Alpha0 code derived from the scheduled Alpha program of FIR filter. This system describes the structure and the behaviour of array shown in Figure 2. In this program, some equations express registers between two Alpha signal of the same processor (e.g. `wPipe[t][p]= ...wPipe[t-1][p]..`) and some equations express registers between two Alpha signals of different

```
for (t = 1; t1 <= N+M-1; ++t1)
  par {
    par (p = 0;p<=0;p++)
        if ((2p+1<=t)&(t<=p+M)) xPipe[t][p] = x[t];
    par  (p = 1; p <= N-1; ++p)
        if ((2p+1<=t)&(t<=p+M)) xPipe[t][p] = xPipe[t-2][p-1];

    par  (p = 0; p <= N-1 ++p)
        par {
            if (t=p+N) wPipe[t][p] = x[t-N];
            if ((p+N+1<=t) & (t<=p+M))  wPipe[t][p] = x[t-N];
            }
    par (p = -1; p <= -1; ++p)
        if ((p+N+1<=t)&(t<=p+M+1)) Y[t][p] = 0;
    par (p = 0; p <= N-1; ++i)
          if ((p+N+1<=t)&(t<=p+M+1))
              Y[t][p] = Y[t-1][p-1]+ wPipe[t-1][p] * xPipe[t-1][p];
}
```

Figure 6: Sketch of the Handel program resulting of the naive translation of the scheduled fir program of figure 3

processors (e.g. `wPipe2[t][p]=...wPipe2[t-2][p-1]..`, there are two registers here). In the translation in Handel of this program, this `wPipe2` signal will give rise to a signal (called `wPipe2`) and two register arrays indexed by `p` (called `wPipe2Reg1[p]` and `wPipe2Reg2[p]`). The selection made by the case branches in the equation defining `wPipe2` can be implemented with a conditional selection. The resulting translation of the equation defining `wPipe2` in HandelC will be (we will explain how we handle the input later):

```
signal int 16 wPipe;
int 16 wPipeReg1[N],wPipeReg2[N];
...
while (True) {
 par (p=0;p<=N-1;p++)
    par { wPipe2= (p==0) ? inputW : ((p>0) ? wPipe2Reg2[p-1] : 0)
          wPipe2Reg2[p]=wPipe2Reg1[p];
          wPipe2Reg1[p]=wPipe2;
      }
```

Note that we do not need an array of signals because we know (as MMAlpha derives systolic architectures) that any connexion between different processors will be associated to a register (there is no broadcast). hence, it is always possible to use signals inside a processor and register in the communications between processors. This fairly simple translation scheme can be applied to all Alpha0 equations except the controler equation (`wPipe_ctl1P_Init` variable). This control variable should be considered as an input to the systolic array and implemented by a simple state machine in Handel activated by the global `start` signal of the design. The resulting Handel program will be composed of a sequential `while (True)` enclosing the body of the program which should take one clock cycle. The body of the program itself is composed of a replicated `par` indexed by the processor index `p` (for more complicated examples there may be several processor indices) . The different behaviours of different processors is selected by a conditional selection.

Finally the input to the systolic design is done with a Handel `channel`. The architecture has to receive two streams of data (`x` and `w`) together with the control signal (`wPipe_ctl1P_Init`). It is up the designer to send the values in the channels in the correct order (this order can be determined by looking in the Alpha0 program to subscript of `w` and `x`).

The architecture (figure 2) will be synthesized with Handel hardware control mechanism (one control latch for each register), but as the body of the program takes only one clock cycle, the Handel compiler can optimize the control and realize that it can just broadcast the clock enable signal to all the registers of the design. This parallel architecture can now easily be incorporated into a more complex design involving control dominant computations since the Handel language has sufficient expressive power for both styles of designs.

4 Combining macro and micro parallelism

The architecture which has been used for illustrating our translation is pipelined at the word level, which is relatively coarse grain. Notably, the implementation of the multipliers is undefined in the resulting abstract architecture. The rate at which the array can be pipelined will be determined by the slowest element of the array, and multiplication can pose a bottleneck. Hence, in order to achieve fast cycle times (computing at the rate of addition) a refinement of the multiplication operations is necessary.

A natural solution is to obtain a pipelined design for the multiplier and this can be realized as a program transformation ahead of hardware compilation. One of the advantages of our approach is to allow the generation of finer grained pipelines as high level transformations in the design process, typically with a range of pipelined multipliers, without increasing the complexity of the design process. The idea here is to express the multiplication as an Alpha program at the boolean level and apply the same transformation steps as in the macro pipeline generation to derive micro pipelines. This idea is particularly relevant in the context of FPGA compilation, because most recent FPGA chips provide specific hardware for fast carry propagation. Therefore, the back-end tool has little choice when implementing additions, and can be expected to do it well. Conversely, the design space for multipliers is quite complex, and requires design choices involving low-level considerations. Our approach is to lift up these design choices to the specification level, and let MMAlpha handle the details of pipeline management.

We have derived a bit level specification of the FIR filter in Alpha (it basically consists in adding one dimension to all variables: the bits and changing operators into calls to library). The script used to derive the word level architecture can be re-used to derive the bit level architecture. The resulting architecture will contain only boolean operations. If the multiplier is completely pipelined, the resulting clock cycle will be not more than that one of an adder thus increasing the throughput of the array.

5 Conclusion

We have explained our methodology which uses two important tools for obtaining highly parallel hardware designs. These hardware designs will also contain a complex control part, and this is the major originality of this work. There are many advantage of Handel over conventional hardware description languages for high level design. Handel (as well as Alpha) has clean semantics allowing high confidence in correctness of translation between Alpha and Handel. Handel provides high level and low level simulation which is very useful for debugging. Finally, the ability to express control-intensive part of application in high level form is very important to provide correct designs.

In this paper, no experimental results are presented yet because this is currently on-going work, but we believe that the idea of collaboration between data parallel model and communicating sequential process model is an important subject that has to be discussed by the scientific community.

References

[1] Api-Cosi. *MMAlpha Reference Manual*, 1997.

[2] E.A. Lee et al. Overview of the ptolemy project. Technical Report UCB/ERL No. M99/37, University of California, Berkeley, july 1999.

[3] A.C. Guillou, P. Quinton, T. Risset, and D. Massicotte. High Level Design of Digital Filters in Mobile Communications. DATE Design Contest 2001, March 2001. Second place, available at http://www.irisa.fr/bibli/publi/pi/2001/1405/1405.html.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[5] Charles E. Leiserson and James B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 1991.

[6] David May. Occam. *Computer Bulletin*, 1(1):14, March 1985.

[7] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI signal processing*, 12:87–107, 1996.

[8] Guy-René Perrin and Alain Darte, editors. *The Data Parallel Programming Model*, volume 1132 of *LNCS Tutorial*. Springer Verlag, 1996.

[9] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, sep 2000.

[10] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of Efficient Nested Loops from Polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[11] J.M. Spivey. Deriving a Hardware Compiler from operational semantics. Technical Report available from http://oldwww.comlab.ox.ac.uk/oucl/users/mike.spivey/, Oxford University Computing Laboratory, 1997.

[12] M. Spivey and I. Page. How to design hardware with Handel. Technical report, Oxford University Hardware Compilation Group, December 1993.

```
system fir :   {N,M | 3<=N<=M-1}
               (x : {n | 1<=n<=M} of integer;
                w : {i | 0<=i<=N-1} of integer)
       returns (res : {n | N<=n<=M} of integer);
var
  -- most declaration omitted
  wPipe : {t,p | p+N<=t<=p+M; 0<=p<=N-1} of integer;
  Y : {t,p | p+N+1<=t<=p+M+1; -1<=p<=N-1} of integer;
let
  wPipe_ctl1P_Init_In[t,p] = wPipe_ctl1P_Init[t];
  wPipe_ctl1P_Init[t] = case  case
          { | t=N} : True[];
          { | N+1<=t} : False[];
        esac;  esac;
  Y1[t,p] = Y[t,p];
  wPipe2[t,p] =  case
        { | p=0} : w[-t+N];
        { | 1<=p} : wPipe2[t-2,p-1];
      esac;
  xPipe[t,p] =   case
        { | p=0} : x[t];
        { | 1<=p} : xPipe[t-2,p-1];
      esac;
  wPipe_ctl1P[t,p] =      case
        { | p=0} : wPipe_ctl1P_Init_In[t,p];
        { | p+N<=t; 1<=p} : wPipe_ctl1P[t-1,p-1];
      esac;
  wPipe_ctl1[t,p] = wPipe_ctl1P;
  wPipe[t,p] =
     case
         { | p+N<=t} : if (wPipe_ctl1) then
               case
                 { | t=p+N} : wPipe2;
                 { | p+N+1<=t} : 0[];
               esac else
               case
                 { | t=p+N} : 0[];
                 { | p+N+1<=t} : wPipe[t-1,p];
               esac;
      esac;
  Y[t,p] =      case
        { | p=-1} : 0[];
        { | 0<=p} : Y[t-1,p-1] + wPipe[t-1,p] * xPipe[t-1,p];
      esac;
  res[n] = Y1[n+N,N-1];
tel;
```

Figure 7: The Alpha0 hardware specification generated from the initial Alpha specification (figure 1). From this specification the architecture of figure 2 can be easily derived in many hardware description languaues.

Here are the informations concerning the paper

the prefered author for contact is :
Tanguy Risset
tel : +33 (0)4 72 72 85 48
fax : +33 (0)4 72 72 80 80 trisset@ens-lyon.fr

Paper title: Design of highly parallel architectures with Alpha and Handel

authors:
Florent Dupont de Dinechin,
INRIA/LIP
ENS-Lyon 46 allée d'Italie 69364 Lyon Cedex 07,
France
fdedinec@ens-lyon.fr

Tanguy Risset
INRIA/LIP
ENS-Lyon 46 allée d'Italie 69364 Lyon Cedex 07,
France
trisset@ens-lyon.

M. Manjunathaiah
OUCL,
Parks Road Oxford, OX1 3QD,
UK
Manju.Manjunathaiah@comlab.ox.ac.

Mike Spivey
OUCL,
Parks Road
Oxford, OX1 3QD,
UK
Mike.Spivey@comlab.ox.ac.uk

All appropriate organizational approvals for the publication of this paper have been obtained. The author(s) will present the paper at the Forum.