DECOMPOSER: A Synthesizer for Systolic Systems

Pao-Po Hou Robert Michael Owens Mary Jane Irwin

Department of Computer Science
The Pennsylvania State University
University Park, PA 16802

ABSTRACT

A tool for synthesizing systolic systems is introduced. Given a hierarchical specification of the computations to be performed and hints as to how these computations are to be performed, this tool generates an analysis of the hardware required to do the computations. The computations are specified as directed acyclic graphs and the hints tell the temporal and topological relationships of each computation. The systolic system is synthesized by traversing the graph and marking each computation with a processor name and a time stamp. This tool, called DECOMPOSER, is the newest entry in a tool set currently under development at Penn State [IO1]. Its output can subsequently be fed to the remaining tools in the tool set to generate a VLSI fabrication description of the systolic system.

Introduction

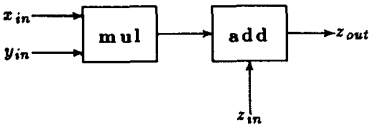
By system synthesis we mean the development of a target architecture from a high level description of an application. In our case, the applications are limited to those computations that are categorized as systolic. We are particularly interested in digital signal processing problems. System synthesis is the first step in the design process; its results can then be fed to module synthesis, module generation, system generation and system validation tools to complete one cycle of the design process. System synthesis is basically a space-time mapping of required computations onto hardware with an assignment of time sequences so that the problem can be solved economically. We refer to this process as *marking* because during this process each computation in the problem is marked with a processor name (where it is to be performed) and a time stamp (when it is to be performed).

As the complexity of VLSI designs grow, system synthesis becomes an increasingly more difficult task. Thus it is fortunate when system synthesis, as with the other steps in the design process, can be automated. The availability of this kind of tool not only simplifies the design task, but, since it allows the designer to quickly generate many different feasible target architectures, a whole spectrum of solutions to the problem can be explored. Since the designer is no longer tied down by the tedious details of implementing a particular solution, he can be more concerned with what solution is best suited to his needs. The final design is then a balance between speed and size as limited by the degree of pipelining and concurrency, where the former determines the maximum utilization of the hardware and the later determines the operation speed. In our architecture, pipelining can take place at both the operand level (via digit serial processing) and the operation level. Our synthesis tool can handle both aspect of pipelining.

User Interface

Our synthesis tool, which we call DECOMPOSER, takes as input a dag (directed acyclic graph) description of the problem and marking information. The dag is a description of the computations to be performed and the marking information tells how they are to be performed. The computation graph is then collapsed upon itself with the guidance of the marking information to produce an implementation graph. Though we have stressed computation mapping in our discussions up to this point, another important task for DECOMPOSER is to determine the data input/output sequence - when and where each piece of data is to be fed into the hardware and retrieved from it. This, together with the hardware description is generated as output.

Problems are invariably developed and specified as algorithms or as mathematical formula. Unfortunately, DECOMPOSER does not take these forms as its input. The algorithms and/or formula have to be transformed into a computation dag. It is generally not hard to transform a mathematical formula into an equivalent dag. For example $z_{out} = z_{in} + x_{in} y_{in}$ can be represented as



Still, this transformation is yet another step in the design process which the designer is burdened with. However, the designer must know the dag description of his computation, otherwise he can not give hints about how to mark it. Future work will try to address this problem.

CH2540-3/88/0000/0650$01.00 © 1988 IEEE

We represent dags in textual form as 1) a set of input and output ports (terminal vertices in the dag), 2) a set of components (interior vertices) and 3) a set of interconnections. Note that a component is itself a dag and if it has no subcomponents (no interior vertices and thus no interconnections), it is a primitive component. The above dag is represented as shown below.

```
interproduct()
{
  ports: xin, yin, zin, zout;
  components: add, mul;
  interconnections: mul.xin = xin, mul.yin = yin,
    add.xin = mul.zout, add.yin = zin, zout = add.zout;
}
```

This is a hierarchical definition. The components *add* and *mul* are themselves dags with the following definitions.

```
mul()
{
  ports: xin, yin, zout;
}

add()
{
  ports: zin, zout;
}
```

Since they do not have interconnection and component clauses in their definitions, they are primitive components. A primitive component can also be interpreted as an operation whose boolean definition is known. A hierarchical dag is reduced to a flat dag, one that contains only primitive components, by way of recursive substitution of each nonprimitive component with its definition with suitable renaming.

The syntax of the interconnection clause needs a little explanation. The clause

```
interconnections: mul.xin = xin;
```

means that port *xin* of subcomponent *mul* is connected to port *xin* of the current component. The clause

```
interconnection: mul.zout = add.xin;
```

means that port *zout* of subcomponent *mul* is connected to port *xin* of subcomponent add. Since connections are bidirectional, it is immaterial as which port appears on the left of the equal sign.

For flexibility, a definition may be parameterized. The following is a partial description of a p digit signed-digit adder [IO2].

```
adder(p)
{
  ports: xin[i], yin[i], zout[i] where 0 ≤ i < p;
  components: add[i], tot[i] where 0 ≤ i < p;
  interconnections: add[i].xin = xin[i], add[i].yin = yin[i],
  ...
}
```

Thus, a single description can be used to describe a family of computation dags.

Marking information is used to mark each primitive component in the flat description with a processor name and a time stamp. It is also used to associate a named piece of input or output data with a port, determine how the processors are to be placed, and determine local storage requirements. Marking rules are given so that an entity is marked according to how its immediate neighbors are marked. The following is a possible marking rule for the primitive component *tot* where *sin, cin* and *zout* are ports on *tot.*

```
tot
{
  internal: p:j = sin(p:j), p:j+1 = cin(p:j);
  external: p:j = zout(z[j]);
}
```

Internal marking information is used to mark components and external marking information is used to mark ports. Thus, the clause

```
internal: p:j+1 = cin(p:j);
```

means that this component should be marked as processor p and given a time stamp $j+1$ provided that the processor connected through port *cin* is marked as p with time stamp j. The clause

```
external: p:j = zout(z[j]);
```

means that if the current processor is marked as $p:j$ then the data associated with port *zout* is *z[j]*. The time stamp may be any integer and the processor name may have subscripts like *p[0]*. To begin the marking process, a special component in the flat description is designated as *distinguished.* A distinguished component's marking information is given without reference to its neighbors. Each connected component of the computation dag must have at least one distinguished component.

A Design Example

The operation of DECOMPOSER is conceptually easy to describe. First, the computation dag description and marking information are read. The user is then queried so that the free variables used to parameterize the computational dag can be assigned a value. The description for each component is unparameterized. The computation dag description is then flattened. The distinguished components are marked. The other components are marked in a depth first manner. Components with the same processor marking are collapsed together. Interconnection consistency checks with regard to the collapsed graph are made. Dangling ports are connected to memories (at present the assumption is made that serial memories (i.e., shift registers) are sufficient for this purpose). The GLUE [IO1] description for each processor type is located in the primitive component GLUE library. The implementation dag is characterized (at present various types of meshes can be characterized). Finally, a description of the implementation dag with its input and output requirements is generated.

For example consider the following script generated by running DECOMPOSER (annotations are delimited from the script by horizontal bars).

Begin program execution.

```
% decomposer time -m warp
```

Produce a listing of the computation dag description.

```
************ COMPUTATION DESCRIPTION ************

***  1 *** /*
***  2 ***  * Time Warp processor.
***  3 ***  */
***  4 *** warp(n, b)
***  5 *** {
***  6 *** ports:          xi[i], yi[i], zo where 0 <= i < n;
***  7 *** distinguished: cell[n - 1, n - 1] = [n - 1];
***  8 *** components:   cell[i, j] where
***  9 ***                  0 <= i < n and 0 <= j < n and
*** 10 ***                  i >= j - b and i <= j + b;
*** 11 *** connections:   cell[i, j].xi = xi[i] where
*** 12 ***                  0 <= i < n and 0 <= j < n and
*** 13 ***                  i >= j - b and i <= j + b;
*** 14 ***                cell[i, j].yi = yi[j] where
*** 15 ***                  0 <= i < n and 0 <= j < n and
*** 16 ***                  i >= j - b and i <= j + b;
*** 17 ***                cell[i, j].d10 = cell[i - 1, j].zo where
*** 18 ***                  1 <= i < n and 0 <= j < n and
*** 19 ***                  i - 1 >= j - b and i <= j + b;
*** 20 ***                cell[i, j].d01 = cell[i, j - 1].zo where
*** 21 ***                  0 <= i < n and 1 <= j < n and
*** 22 ***                  i >= j - b and i <= j - 1 + b;
*** 23 ***                cell[i, j].d11 = cell[i - 1, j - 1].zo
*** 24 ***                  where 1 <= i < n and 1 <= j < n
*** 25 ***                  and i >= j - b and i <= j + b;
*** 26 *** }
*** 27 ***  /*
*** 28 ***   * Comparer cell.
*** 29 ***   */
*** 30 ***  cell()
*** 31 ***  {
*** 32 ***  ports:    xi, yi, zo, d01, d10, d11;
*** 33 ***  }
```

Produce a listing of the marking information.

```
*************** PAINT DESCRIPTION ***************

***  1 *** cell {
***  2 *** internal:       q[i][a, b]:1 = d10(p[i + 1][a - 1, b]:1 - 2);
***  3 ***                 q[i][a, b]:1 = d01(p[i][a, b - 1]:1 - 2);
***  4 ***                 q[i][a, b]:1 = d11(q[i][a - 1, b - 1]:1 - 4);
***  5 ***                 p[i][a, b]:1 = d10(q[i][a - 1, b]:1 - 2);
***  6 ***                 p[i][a, b]:1 = d01(q[i - 1][a, b - 1]:1 - 2);
***  7 ***                 p[i][a, b]:1 = d11(p[i][a - 1, b - 1]:1 - 4);
***  8 *** external:       q[i][a, b]:1 = xi(xi[a]);
***  9 ***                 q[i][a, b]:1 = yi(yi[b]);
*** 10 ***                 p[i][a, b]:1 = xi(xi[a]);
*** 11 ***                 p[i][a, b]:1 = yi(yi[b]);
*** 12 *** distinguished: p[0][i, i]:0 = [i];
*** 13 *** }
```

Query the user for parameterization data (the sample size is n and the warp factor is b).

```
parameter n? 6
parameter b? 2
```

Output the physical description for each collapsed component.

```
************** PHYSICAL DESCRIPTION **************
  p(i3, i2, o0, i1, i0)
  {
       cell(i0, i1, o0, i2, i3, o0);
  }
  q(i3, i2, o0, i1, i0)
  {
       cell(i0, i1, o0, i2, i3, o0);
  }
```

Output the implementation dag description and input/output information.

```
*************** MODULE DESCRIPTION ***************

  i4 -> yi[2]:-16 yi[3]:-12 yi[4]:-8 yi[5]:-4
  i7 -> xi[2]:-16 xi[3]:-12 xi[4]:-8 xi[5]:-4
  i0 -> yi[1]:-18 yi[2]:-14 yi[3]:-10 yi[4]:-6 yi[5]:-2
  i3 -> xi[1]:-18 xi[2]:-14 xi[3]:-10 xi[4]:-6 xi[5]:-2
  i6 -> yi[0]:-16 yi[1]:-12 yi[2]:-8 yi[3]:-4
  i2 -> yi[0]:-18 yi[1]:-14 yi[2]:-10 yi[3]:-6 yi[4]:-2
  i8 -> yi[0]:-20 yi[1]:-16 yi[2]:-12 yi[3]:-8 yi[4]:-4 yi[5]:0
  i5 -> xi[0]:-16 xi[1]:-12 xi[2]:-8 xi[3]:-4
  i1 -> xi[0]:-18 xi[1]:-14 xi[2]:-10 xi[3]:-6 xi[4]:-2
  i9 -> xi[0]:-20 xi[1]:-16 xi[2]:-12 xi[3]:-8 xi[4]:-4 xi[5]:0

  warp(i4,i7,i0,i3,i6,i2,i8,i5,i1,i9)

    {
         q[0](t0,t1,t2,i0,i1);
         q[-1](t1,t3,t4,i2,i3);
         p[1](0,t2,t0,i4,i5);
         p[-1](t4,0,t3,i6,i7);
         p[0](t2,t4,t1,i8,i9);
    }

     p[1]          p[0]          p[-1]

          q[0]          q[-1]
```

Figure 1 gives the flattened computation dag for the example above and Figure 2 gives the implementation dag.

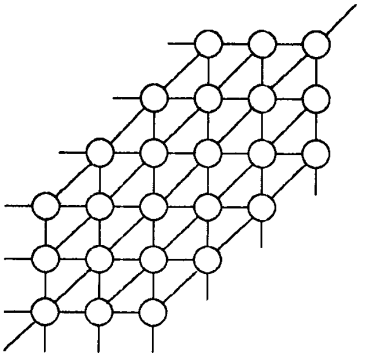


Figure 1. Flattened Computation DAG

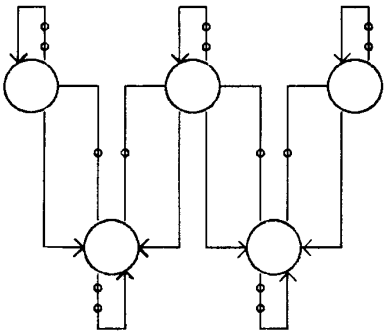


Figure 2. Implementation DAG

The VLSI layout for each collapsed component can be created using the physical description generated by DECOMPOSER for that component and the GLUE descriptions for the primitive components stored in the GLUE library. With the implementation dag and the GLUE description for each collapsed component, the other tools in the tool set can produce a VLSI layout description for the target architecture.

Comparisons

In recent years, system synthesis has received moderate attention. Most of the methods proposed to automate system synthesis [CaS, Ch1, Ch2, LiW] involve using linear transformations to map an n dimensional homogeneous structure onto an $n-1$ dimensional hyperplane. Such a method applies only to a restrict class of systolic algorithms. Advantages of our approach over other systolic mapping schemes are discussed below.

Our approach can handle nonrectangular descriptions. Approaches which have been suggested by others use, for example, planes on rectangles [CaS, Ch1, Ch2] and are therefore tied to computation descriptions which can viewed in terms of rectangular meshs. Also, their use of data dependence vectors to capture the structure of the computation is obviously weaker than using a dag description.

Our approach can handle local storage. Other approaches which use just (in a sense) a description of the computation to be performed have problems with local storage since the description of the computation usually lacks information about such things. Our marking description supplies us with this information.

Our approach can handle systems which have multiple operation modes, for example a distinct loading mode and executing mode. When local storage is involved, a separate loading mode may be required to initialize the state of the memories. Again, our marking information supplies such information.

Conclusions

This paper has presented a system synthesis tool which is compatible with the rest of our tool set. The main strength of our system synthesis approach comes from a more expressive computation description. This description, while being more expressive, does not add significantly to the burden placed on the designer. Armed with the entire tool set [IO1], the designer supplies the appropriate computation description and obtains a, possibly, multi-chip VLSI layout description.

DECOMPOSER has been used to assist in the evaluation of the design of several systolic systems including the Arithmetic Cube [OI1], a dynamic time warp speech processor [Ir], a dynamic space warper for image processing, a Lattice filter, and various other signal processors.

Acknowledgements

This work was supported in part by the U.S. Army Research Office under Contract No. DAAL03-87-K-0118 and the National Science Foundation under Grant No. MIP-8701367.

References

CaS Cappello, P. and K. Steiglitz, Unifying VLSI Array Designs with Geometric Transformations, *Proc. of Inter. Conf. on Parallel Proc.*, pp 448-457, 1983.

Ch1 Chen, M., The Generation of a Class of Multipliers: Synthesizing Highly Parallel Algorithms in VLSI, *IEEE Trans. on Computers*, C-37(3), pp 329-338, March 1988.

Ch2 Chen, M., A Design Methodology for Synthesizing Parallel Algorithms and Architectures, *Journal Parallel Dist. Computing*, Dec. 1986.

Ir Irwin, M.J., A Digit Pipelined Dynamic Time Warp Processor, CS-86-23, PSU, August 1986, to appear in *IEEE Trans. on ASSP*.

IO1 Irwin, M.J. and R.M. Owens, An Overview of the Penn State Design System, *Proc. of DAC '87*, pp 516-522, July 1987.

IO2 Irwin, M.J. and R.M. Owens, Digit Pipelined Arithmetic as Illustrated by the Paste-Up System: A Tutorial, *IEEE Computer*, 20(4), pp 61-73, April 1987.

LiW Li, G. and B. Wah, The Design of Optimal Systolic Arrays, *IEEE Trans. on Computing*, C-34(1), pp 66-77, Jan. 1985.

OI1 Owens, R.M. and M.J. Irwin, The Arithmetic Cube, *IEEE Trans. on Computers*, C-36(11), pp 1342-1348, Nov. 1987.

OI2 Owens, R.M. and M.J. Irwin, Being Stingy with Multipliers, CS-87-33, PSU, Nov. 1987.

Qu Quinton, P., Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations, *Proc. 11 Symp. Computer Arch.*, pp 208-214, May 1984.