A FORMAL DESIGN METHODOLOGY
FOR PARALLEL ARCHITECTURES

KHALED M. ELLEITHY & MAGDY A. BAYOUMI
The Center For Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504, U.S.A.

ABSTRACT-- In this paper, we introduce a formal approach for synthesis of array architectures. Four different forms are used to express the input algorithm: simultaneous recursion, recursion with respect to different variables, fixed nesting and variable nesting. Four different architectures for the same algorithm are obtained. As an example, a matrix-matrix multiplication algorithm is used to obtain four different optimal architectures. The different architectures of this example are compared in terms of area, time, broadcasting and required hardware.

1. INTRODUCTION

Reported techniques for high level synthesis suffer from the following disadvantages: some systems are not suitable for large problems (Emerlad[1]), some systems require to know the target architecture in advance in order to have the structural description [2-4], a number of restrictions are imposed on the input description (Flamel[5], HARP[6]), Certain phases of the design process are not automated due to the lack of algorithms to transform between different representations; such limitations require the designer's responsibility for designing these phases (HARP[6]), the designer is responsible for specifying the operations sequencing and communications among different units [7,8], and some systems are limited to a special class of algorithms [9-11].

A formal design methodology for high level synthesis has been introduced in [12-14] to overcome the previous disadvantages. The architectures produced by this methodology can be classified as uniprocessor architectures. To exploit the parallelism in a given algorithm the methodology has been generalized so that it can be applied to the simultaneous recursion form [15,16]. In this paper the methodology is applied to the following forms:

1. Recursion with respect to several variables.
2. Fixed nested recursion.
3. Variable nested recursion.

The methodology provides two main features: completeness and correctness. Completeness means the ability to use the approach for any general algorithm. Correctness is achieved by using a set of transformations that are proved to be correct. A formal framework for the synthesis procedure has been developed which can be easily automated. A design example of matrix-matrix multiplication is used with each one of the forms to obtain a parallel architecture. These different architectures for this
example are compared in terms of area and speed.

2. RECURSION WITH RESPECT TO SEVERAL VARIABLES

If $x_i$ ($1 \leq i \leq n$) are $n-1$ place functions, $z$ is $n$ place functions, $y$ is $2n$ place function and $w^{(j)}_i$ are $n$ place functions, then $z$ is defined by the following Algorithm Specification Language (ASL) code:

\begin{align*}
  z(0, arg_3, \ldots, arg_n) &= x_1(arg_3, \ldots, arg_n) \\
  z(arg_1+1, 0, arg_3, \ldots, arg_n) &= x_1(arg_3, \ldots, arg_n) \\
  \vdots
\end{align*}

\begin{align*}
  z(arg_1+1, arg_3+1, \ldots, arg_{n-1}+1, 0) &= x_n(arg_1, \ldots, arg_{n-1}) \\
  z(arg_1+1, \ldots, arg_{n}+1) &= y(arg_1, \ldots, arg_n, z_1, \ldots, z_n) \quad (2)
\end{align*}

\begin{align*}
  z_1 &= z(arg_1, w_1^{(1)}, w_2^{(1)}, \ldots, w_{n-1}^{(1)}) \\
  z_2 &= z(arg_1+1, arg_3, w_1^{(2)}, w_2^{(2)}, \ldots, w_{n-2}^{(2)}) \\
  \vdots
\end{align*}

\begin{align*}
  z_{n-1} &= z(arg_1+1, \ldots, arg_{n-1}+1, arg_{n-1}, w_1^{(n-1)}) \\
  z_n &= z(arg_1+1, \ldots, arg_{n-1}+1, arg_n) \quad (3)
\end{align*}

**Transformation Algorithm to RSL**

To transform the system of recurrence with respect to several variables to the Realization Specification Language RSL representation we implement each equation using the same method described in (12-14). Here is the RSL representation of the system:

\begin{align*}
  \text{Initp}(1, arg_1; 2, arg_2; \ldots; n, arg_n) \\
  \text{succ}(\text{control}_1) &= x_1(arg_2)^{\text{ready}} \quad (1)
\end{align*}

\begin{align*}
  \text{succ}(\text{control}_1) &= x_1(arg_2)^{\text{ready}} \quad (2)
\end{align*}

\begin{align*}
  \text{succ}(\text{control}_2) &= x_1(arg_2)^{\text{ready}} \quad (3)
\end{align*}

\begin{align*}
  I &= p_1^{\text{succ}(I)} \quad (4)
\end{align*}

\begin{align*}
  \text{Ready} &= eq(I, m) \quad (5)
\end{align*}

\begin{align*}
  z(0, arg_3, \ldots, arg_n) &= \text{Comp}(arg_3, \ldots, arg_n, \# x_1) \\
  z(\text{Ready}, 0, arg_3, \ldots, arg_n) &= \text{And}(\text{arg}_{\text{Ready}}, \ldots, \text{arg}_{\text{Ready}}) \quad (6)
\end{align*}
\[ z \left( \text{arg}_1 + 1, 0, \text{arg}_3, \ldots, \text{arg}_n \right) = \text{Comp} \left( \text{arg}_1 + 1, 0, \text{arg}_3, \ldots, \text{arg}_n \right) \]

\[ z \text{ Ready} \left( \text{arg}_1 + 1, 0, \text{arg}_3, \ldots, \text{arg}_n \right) = \text{And} \left( \text{arg}_1 + 1 \text{ Ready}, \text{arg}_3 \text{ Ready}, \ldots, \text{arg}_n \text{ Ready} \right) \]

\[ z \left( \text{arg}_1 + 1, \text{arg}_2 + 1, \ldots, \text{arg}_{n-1}, 0 \right) = \text{Comp} \left( \text{arg}_1 + 1, \text{arg}_2 + 1, \ldots, \text{arg}_{n-1} + 1, 0 \right) \]

\[ z \text{ Ready} \left( \text{arg}_1 + 1, \text{arg}_2 + 1, \ldots, \text{arg}_{n-1} + 1 \right) = \text{And} \left( \text{arg}_1 + 1 \text{ Ready}, \text{arg}_2 + 1 \text{ Ready}, \ldots, \text{arg}_{n-1} + 1 \text{ Ready} \right) \]

\[ z \left( \text{arg}_1 + 1, \text{arg}_2 + 1, \ldots, \text{arg}_n + 1 \right) = \text{Comp} \left( \text{arg}_1 + 1, \text{arg}_2 + 1, \ldots, \text{arg}_n + 1, z_1, \ldots, z_n \neq y \right) \]

\[ z \text{ Ready} \left( \text{arg}_1 + 1, \text{arg}_2 + 1, \ldots, \text{arg}_n + 1 \right) = \text{And} \left( \text{arg}_1 + 1 \text{ Ready}, \text{arg}_2 + 1 \text{ Ready}, \ldots, \text{arg}_n + 1 \text{ Ready} \right) \]

\[ z_1 = \text{Comp} \left( \text{arg}_1, w_{1(1)}, \ldots, w_{n-1(1)} \neq z \right) \]

\[ z_1 \text{ Ready} = \text{And} \left( \text{arg}_1, w_{1(1)} \text{ Ready}, \ldots, w_{n-1(1)} \text{ Ready} \right) \]

\[ z_2 = \text{Comp} \left( \text{arg}_2 + 1, \text{arg}_2 w_{2(2)}, \ldots, w_{n-2(2)} \neq z \right) \]

\[ z_2 \text{ Ready} = \text{And} \left( \text{arg}_2 + 1 \text{ Ready}, \text{arg}_2 w_{2(2)} \text{ Ready}, \ldots, w_{n-2(2)} \text{ Ready} \right) \]

\[ z_{n-1} = \text{Comp} \left( \text{arg}_1 + 1, \ldots, \text{arg}_n + 1, \text{arg}_{n-1} + 1, w_{1(n-1)} \neq z \right) \]

\[ z_{n-1} \text{ Ready} = \text{And} \left( \text{arg}_1 + 1 \text{ Ready}, \ldots, \text{arg}_{n-1} + 1 \text{ Ready}, \text{arg}_{n-1} w_{1(n-1)} \text{ Ready} \right) \]

\[ z_n = \text{Comp} \left( \text{arg}_1 + 1, \ldots, \text{arg}_{n-1} + 1, \text{arg}_n \neq z \right) \]

\[ z_n \text{ Ready} = \text{And} \left( \text{arg}_1 + 1 \text{ Ready}, \ldots, \text{arg}_{n-1} + 1 \text{ Ready}, \text{arg}_n \text{ Ready} \right) \]

Equation 1 is used to show that we use \( n \) registers to be initialized with the arguments \( \text{arg}_1, \ldots, \text{arg}_n \). Equation 2 means that the unit \( \text{Suc} \) which is a basic function has its inputs \( \text{control} \left( 1 \leq i \leq r \right) \) connected to the \( \text{ready} \) output of the unit computing \( x_i \) to be sure that \( I \) is not incremented until \( x_i \) is computed. Equation 3 is used to represent the fact that \( I \) is incremented every cycle using the \( \text{Suc} \) unit, and \( I \) is initialized to the value 1 using the register number \( n+1 \). Equation 4 determines the end of operation when \( I \) reaches the value \( m \). Equations 5, 6, 7 represent the composition operation in equations 1, 2, 3 of the ASL representation respectively. Proof of correctness for the algorithm can be found in [17].

Example:

Let us use the recursion with several variable to define the greatest common divisor (GCD). Two variables are required to define the GCD. The definition is as follows:

\[ \text{GCD} \left( 0, n \right) = n \]
\[ \text{GCD} \left( m + 1, 0 \right) = m + 1 \]
\[ \text{GCD} \left( m + 1, n + 1 \right) = \beta \left( m, n, \text{GCD} \left( w \left( m, n \right), n + 1 \right), \text{GCD} \left( m + 1, w \left( m, n \right) \right) \right) \]
\[ w \left( m, n \right) < m + 1 \]
\[ w \left( m, n \right) < n + 1 \]
\[ \beta \left( m, n, \beta \right) = a \quad n > m \]
This is expressed in ASL as follows:

\[ \text{GCD}(0, n) = \pi_2^1(0, n) \]  
(1)

\[ \text{GCD}(m + 1, 0) = \pi_1^2(m + 1, 0) \]  
(2)

\[ \text{GCD}(m + 1, n + 1) = \beta(m, n, \text{GCD}(w(m, n), n + 1), \text{GCD}(m + 1, w(n, m))) \]  
(3)

\[ w(m, n) = \text{less than}(\lambda(\pi_1^2(m, n))) \]  
(4)

\[ w(n, m) = \text{less than}(\lambda(\pi_1^2(m, m))) \]  
(5)

\[ \beta(m, n, a, b) = \text{compare}(m, n, a, b) \]  
(6)

The RSL representation of this example is as follows:

\[ \text{Init}(1, m ; 2, n) \]  
(1)

\[ \text{succ}_1 = \pi_2^1(\text{READY}) \]  
(2)

\[ \text{succ}_2 = \pi_1^2(\text{READY}) \]  
(3)

\[ \text{Ready} = \text{eq}^2(1, m) \]  
(4)

\[ \text{GCD}(0, n) = \text{Comp}(0, n \neq \pi_2^1) \]  
(5)

\[ \text{GCD}_\text{Ready}(0, n) = \pi_1^2 \]  
(6)

\[ \text{GCD}(m + 1, 0) = \text{Comp}(m + 1, 0 \neq \pi_1^2) \]  
(7)

\[ \text{GCD}_\text{Ready}(m + 1, 0) = \pi_1^2 \]  
(8)

\[ \text{GCD}(m + 1, n + 1) = \text{Comp}(m + 1, n + 1, \text{GCD}(w(m, n), n + 1), \text{GCD}(m + 1, w(n, m)) \neq \beta) \]  
(9)

\[ \text{GCD}_\text{Ready}(m + 1, n + 1) = \text{And}(\pi_1^2, \pi_2^1, \text{GCD}_\text{Ready}(w(m, n), n + 1), \text{GCD}_\text{Ready}(m + 1, w(n, m))) \]  
(10)

\[ w(m, n) = \text{Comp}(\lambda(\pi_1^2(m, n)) \neq \text{less than}(\text{Ready}) \]  
(11)

\[ w(n, m) = \text{Comp}(\lambda(\pi_1^2(m, m)) \neq \text{less than}(\text{Ready}) \]  
(12)

\[ w(m, n) = \text{less than}(\text{Ready}) \]  
(13)

\[ w(n, m) = \text{less than}(\text{Ready}) \]  
(14)

\[ \beta(m, n, a, b) = \text{compare}(m, n, a, b) \]  
(15)
We see in this example the correspondence between ASL and RSL representations. Equation 1 shows that we use 2 registers to be initialized with the arguments \( m, m \). Equation 2,3,4 has the same meaning as equations 2,3,4 in the transformation algorithm. Equations 5,6,7 represent the composition operation in equations 1,2,3 of the ASL representation respectively.

3. NESTED RECURSION

Two cases are considered for the nested recursion: fixed number of nestings and varied number of nestings. Instead of addressing a recursion of \( k \)-fold nesting, we show the idea in every case with a double nesting. The general case of \( k \)-fold can be easily extended.

3.1 Fixed Number of Nestings

If \( x \) is 1-place function, \( z \) is 2-place function, and \( y \) is 2-place function and \( w \) is 3-place function, then \( y \) is a double nested recursion function defined by the following ASL:

\[
\begin{align*}
y(0, a) &= x(a) \\
y(n+1, a) &= z(n, y(n, w(n, a, y(n, a))))
\end{align*}
\]

To see that the previous definition has a fixed number of recursion let us compute the different values for \( y \).

\[
\begin{align*}
y(0, a) &= x(a) \\
y(1, a) &= z(0, y(0, w(0, a, y(0, a)))) \\
&= z(0, y(0, w(0, a, x(a)))) \\
&= z(0, x(w(0, a, x(a)))) \\
y(2, a) &= z(1, y(1, w(1, a, x(1, a)))) \\
&= 1, y(1, w(1, a, x(1, a)))) \\
&= z(1, x(0, x(w(0, a, x(0, x(w(0, a, x(a)))))))) \\
&= z(1, x(0, x(x(0, z(x(0, z(x(a)))))))) \\
&= z(1, x(0, x(x(0, z(x(0, x(a)))))))
\end{align*}
\]

Although the solution of the function \( y \) gets more complicated as we go for higher values, we are still able to solve it in the same previous method. The solution we obtain for \( y(2, a) \) is expressed in the functions \( z, w, x \) but not the function \( y \). This means if the functions \( z, w, x \) are primitive recursive than \( y \) is primitive recursive since it is driven from \( z, w, x \) by composition.

Transformation Algorithm to RSL

To transform the system of nested recursion to RSL we implement each equation using the same method described in [12-14]. Here is the RSL representation of the double nested recursion:

\[Initp(1, a; 2, a)\]

(1)
Proof of correctness for the algorithm can be found in [17].

3.2 Variable Number of Nestings

If $x$ is 1-place function, and $y$ is 2-place function then $y$ is a double nested recursion function defined by the following RSL:

$y(0,n) = x(n)$
$y(m+1,0) = y(m,1)$
$y(m+1,n+1) = y(m,y(m+1,n))$

To see that the previous definition doesn't behave in the same manner as the definition in section 3.1, let us compute values of $y$ for different $m,n$.

$y(0,n) = x(n)$
$y(1,1) = y(0,y(1,0)) = y(0,y(0,1)) = y(0,x(1))$
$y(2,3) = y(1,y(2,2)) = y(1,y(1,y(2,1)))$

From the computation of $y$ for different $m,n$ we notice that the number of nestings is not constant and depends on earlier values. This means that nested recursion can leads out of primitive recursion.

Transformation Algorithm to RSL

To transform the system of nested recursion to RSL we implement each equation using the same method described in [12-14]. Here is the RSL representation of
the double nested recursion:

\[
\text{Init}(1, n, 2, m) \quad (1)
\]
\[
\text{succ}_{\text{control}} = x(n) \quad (2)
\]
\[
I = \rho_{I} \text{ succ }(I_{2}) \quad (3)
\]
\[
I = \rho_{I} \text{ succ }(I_{2}) \quad (4)
\]
\[
\text{Ready} = \text{ And } (eq(I_{1}, n), eq(I_{2}, n)) \quad (5)
\]
\[
y(0, n) = \text{ Comp } (n \neq x) \quad (6)
\]
\[
y_{\text{Ready}}(0, n) = y_{\text{Ready}}(n) \quad (7)
\]
\[
y(m+1, 0) = \text{ Comp } (m, 1 \neq y) \quad (8)
\]
\[
y_{\text{Ready}}(m+1, 0) = y_{\text{Ready}}(m, 1) \quad (9)
\]
\[
\text{Temp} = \text{ Comp } (m+1, n \neq y) \quad (10)
\]
\[
\text{Temp}_{\text{Ready}} = \text{ And } (\rho_{1}, \rho_{2}) \quad (11)
\]
\[
y = \text{ Comp } (m, a, \text{ Temp}_{1} \neq y) \quad (12)
\]
\[
y_{\text{Ready}} = \text{ And } (\rho_{3}^{\text{Ready}}, \text{ Temp}_{3}^{\text{Ready}}) \quad (13)
\]
\[
\text{Temp}_{3}^{\text{Ready}} = \text{ And } (\text{ Temp}_{3}^{\text{Ready}}, \text{ Temp}_{2}^{\text{Ready}}) \quad (14)
\]
\[
y = \text{ Comp } (n, a, \text{ Temp}_{3} \neq x) \quad (15)
\]
\[
y_{\text{Ready}} = \text{ And } (\rho_{1}^{\text{Ready}}, \rho_{2}^{\text{Ready}}, \text{ Temp}_{3}^{\text{Ready}}) \quad (16)
\]

Proof of correctness for the algorithm can be found in [17].

Table 1 shows a comparison between different forms of recursion in terms of architecture, broadcasting and complexity of the controller. The simultaneous recursion is the only form that gives a two dimensional array. All forms have broadcasting except the variable nesting. The controller of the variable nesting is complex compared with the other three forms.

Table 1. Comparison Between Different Forms of Recursion.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Simultaneous</th>
<th>Several variables</th>
<th>Fixed nesting</th>
<th>Variable nesting</th>
</tr>
</thead>
<tbody>
<tr>
<td>two dimensional</td>
<td>one dimensional</td>
<td>one dimensional</td>
<td>one dimensional</td>
<td>complex</td>
</tr>
<tr>
<td>Broadcasting</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Complexity of controller</td>
<td>simple</td>
<td>simple</td>
<td>simple</td>
<td>complex</td>
</tr>
</tbody>
</table>
5. Matrix-Matrix Multiplication Example

An example of matrix multiplication is introduced as an application of different forms of recursion. The architecture has two matrices $A$ and $B$ as inputs, and matrix $C$ as an output. The multiplication is done in a recursive way and can be described by the following high level subroutine:

```plaintext
mat-*multiplication (A,B)
begin
for i=1 to n
    begin
    for j=1 to n
        begin
        $C_{i,j} = 0$
        for k=1 to n
            $C_{i,j,k} = C_{i,j,k-1} + A_{i,k} \times B_{k,j}$
        end
    end
end
```

The ASL and RSL representation using the simultaneous recursion form is as follows:

**ASL Representation**

```
C_{1,1}(A_{1,k}, B_{1,j}, 0) = \xi()
..................................
C_{1,n}(A_{1,k}, B_{1,j}, 0) = \xi()
C_{1,1}(A_{1,k}, B_{1,j}, K) = inner_product(A_{1,k}, B_{1,j}, C_{1,1}(A_{1,k}, B_{1,j}, K-1))
..................................
C_{n,n}(A_{1,k}, B_{1,j}, K) = inner_product(A_{1,k}, B_{1,j}, C_{n,n}(A_{1,k}, B_{1,j}, K-1))
```

**RSL Representation**

```
Init (0,n; 1,A_{1,1}; ...; n^2 A_{n,n}; n^2+1,B_{1,1}; ...; 2n^2 B_{n,n})

 succ_controll = \xi() \emptyset
..................................
 succ_con troll_2 = \xi() \emptyset
 I = \rho_{i} \emptyset
 succ (I)
 Ready = eqt(I, m)

 Result_1 = comp (diff I, n^2+1) Result \# inner_product
..................................
 Result_1 = comp (diff I, n^2+1) Result \# inner_product
```

Figure 1 shows the architecture obtained for matrix multiplication. The details of implementing the inner-product cell are shown in [14]. The architecture consists of $N^2$ inner-product cells. The number of cycles required to perform the multiplication...
Figure 2 shows the architecture using recursion with respect to several variables. The architecture consists of $N$ multiplication cells and one adder. The number of cycles required to perform the multiplication is $N^2$.

Figure 3 shows the architecture using fixed nesting recursion. The architecture consists of $N^2$ inner-product cells. The number of cycles required to perform the multiplication is $N^2$.

Table 2 shows a comparison between different architectures of the matrix-matrix multiplication.

Fig. 1. Matrix-Matrix Multiplication Architecture Using Simultaneous Recursion.
Table 2. Comparison Between Different Matrix-Matrix Multiplication Architectures.

<table>
<thead>
<tr>
<th></th>
<th>Simultaneous</th>
<th>Several variables</th>
<th>Nesting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Area</strong></td>
<td>$\theta(n^3)$</td>
<td>$\theta(n)$</td>
<td>$\theta(n)$</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>$\theta(n)$</td>
<td>$\theta(n^3)$</td>
<td>$\theta(n^2)$</td>
</tr>
<tr>
<td><strong>A*T</strong></td>
<td>$\theta(n^3)$</td>
<td>$\theta(n^3)$</td>
<td>$\theta(n^2)$</td>
</tr>
<tr>
<td><strong>Broadcasting</strong></td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td><strong>Hardware</strong></td>
<td>inner-product</td>
<td>multiplier ($n$)</td>
<td>inner-product</td>
</tr>
<tr>
<td></td>
<td>($n^2$)</td>
<td>adder (1)</td>
<td>($n$)</td>
</tr>
</tbody>
</table>

Fig. 2. Matrix-Matrix Multiplication Architecture Using Recursion With Respect to Several Variables.
6. CONCLUSIONS

In this paper an formal approach for transforming different forms of recursion to parallel architectures has been introduced. Four different forms are used to express a given algorithm. Four optimal architectures for a matrix-matrix multiplication are compared. The approach has the following advantages:

1. It is suitable for large problems since the transformation algorithm is linear.
2. It does not require to know the target architecture in advance.
3. There are no restrictions imposed on the input description.
4. The technique is fully automated.
5. The designer is not responsible for specifying the operations sequencing and communications among different units.
6. The approach is applicable to any general algorithm.
7. Parallel properties of algorithms are explored.

![Diagram of Matrix-Matrix Multiplication Using Fixed Number of Nestings](image)

**Fig. 3.** Matrix-Matrix Multiplication Using Fixed Number of Nestings.
References