Proceedings of the 10th WSEAS International Conference on CIRCUITS, Vouliagmeni, Athens, Greece, July 10-12, 2006 (pp75-80)

Solving the Maximum Subsequence Problem with a Hardware Agents-based System

Octavian Creț[1], Zsolt Mathe[1], Cristina Grama[1], Lucia Văcariu[1], Flaviu Roman[1], Adrian Dărăbant[2]
Computer Science Department
[1] Technical University of Cluj-Napoca; [2] "Babeş-Bolyai" University of Cluj-Napoca
26, G. Barițiu Street, Cluj-Napoca; 1, M. Kogălniceanu Street, Cluj-Napoca
ROMANIA

Abstract – The maximum subsequence problem is widely encountered in various digital processing systems. Given a stream of both positive and negative integers, it consists of determining the subsequence of maximal sum inside the input stream. In its two-dimensional version, the input is an array of both positive and negative integers, and the problem consists of determining the sub-array of maximal sum inside the input array. These problems are solved by Kadane's algorithm, which has already been proved to be optimal. However, the hardware implementation presented in this paper is based on the hardware agents' paradigm and offers a significantly improved performance (in terms of speed) over the classical software implementations.

Key-words: Kadane algorithm, maximal subsequence problem, hardware agents, VHDL, FPGA

1 Introduction

The problem of maximum sum was first introduced by Bentley [1, 2] and has 1D and 2D versions. It is not very difficult to implement in software. The 1D version is referred to as the maximum subsequence, while the other is called the maximum sub-array problem. It is implemented for various DSP applications [6]; Bentley [1] introduced Kadane's algorithm that finds the maximal subsequence of a 1D array in linear time $O(n)$.

The maximum subsequence problem is very widely encountered in various digital processing systems. It consists of determining a contiguous portion of an unsorted 1D array that contains both positive and negative integers in which the sum of the array elements is maximal, over the entire stream.

For instance, in the following 13-element 1D array:

$$a[13] = \{7, -9, 15, 20, -37, 23, 4, 5, 19, -28, 17, -2, 1\}$$

If we consider the first element to be $a[0]$, the maximal sub-array is delimited by the element of index 5 ($a[5] = 23$) and the element of index 8 ($a[8] = 19$).

In the following, let s be the notation for the sum of a tentative maximum subsequence $a[i..j]$. The algorithm accumulates a partial sum in t and, when t becomes better than s, it replaces s with t and updates the position (i, j). If t becomes negative, we reset the accumulation.

This operation is due to its dynamic programming nature; a subsequence of negative sum can only constitute a suboptimal maximum sum, thus it is discarded. The algorithm is presented in Fig. 1.

```
1.     input : a[1..n]
2.     output : s,(x1, x2)
3.     (x1, x2) ← (0, 0); s ← – ∞; t ← 0;
4.     i ← 1;
5.     for j ← 1 to n do
6.     t ← t +a[j] ;
7.         if t > s then
8.     (x1,  x2) ← (i, j); s ← t
9.         end
//Reset the accumulation
10.        if t < 0 then
11.               t ← 0; i ← j + 1
12.        end
13.    end
```

Fig. 1. Kadane's algorithm

One of the major tendencies in digital computing systems is to relocate the computation-intensive parts of applications into hardware. Processors have a general, fixed architecture that allows the implementing of tasks by *temporally* composing atomic operations provided e. g. by the ALU or the floating-point unit. In contrast, ASICs implement tasks by *spatially* composing operations provided by dedicated functional units. Reconfigurable computing combines both approaches. Reconfigurable systems are built from programmable logic, which allows the implementing of tasks both in a spatial manner similar to ASICs and in a temporal manner comparable to processors.

The main goal of our research group is to develop hardware implementations for the most widely used classical software algorithms. Thus, it will be possible to gradually implement larger systems based on specialized cooperative hardware agents, where each agent is responsible for implementing a given algorithm.

Proceedings of the 10th WSEAS International Conference on CIRCUITS, Vouliagmeni, Athens, Greece, July 10-12, 2006 (pp75-80)

This paper shows the results for the implementation of Kadane's algorithm in software and in hardware, focusing on the hardware implementations which yield much better results. The design technique is detailed, and the necessary adaptations are explained. Finally, the architecture is loaded into a Xilinx FPGA device, for simulation and testing purposes.

2 The 1D Implementation

The software implementation for the 1D case is straightforward and does not deserve to be detailed here. The program was written in C++ and tested on a few PC systems. The obtained results are presented in Table 1.

Table 1. Results of the software (C++) implementation

Number of Tests	Input Stream Length	Run time (sec)
10	10000	0.16
10	20000	0.3
10	30000	1.012
100	10000	1.722
100	20000	3.385
100	30000	5.208
1000	10000	17.805
1000	20000	37.013
1000	30000	45.956
1000	60000	93.789

These results are due to the *temporal* organization of the computing task in the microprocessor-based architecture. They are also limited by the numerous memory accesses required by this algorithm.

As it can be seen in Table 1, the run time for a sequence of 60,000 numbers is of approximately 90 ms.

The much better results achieved by the hardware implementation are due to the spatial organization of the computing task in an FPGA device, which has been used as physical support for the implementation.

When moving it to hardware, the algorithm must first be adapted. We consider the case of the analyzed stream received serially, synchronously, on our system's data input. The outputs generated by the design are the two pointers to the beginning and the end of the maximal sequence, x_1 and x_2, and the maximal sum obtained, s.

As Naji & Wells have shown in [4], hardware multi-agent systems allow the concepts of multi agent technology to be extended to support reconfigurable systems, in which the functionality of both the associated hardware and software can be altered dynamically or statically with time. The utilization of such a paradigm has the potential to greatly increase the flexibility, efficiency, expandability, and maintainability of such systems and to provide an attractive alternative to the current set of disjoint approaches that are now applied to this problem domain.

This paper presents an example of how an agents-based architecture can be created and applied to the reconfigurable hardware domain. We'll consider the system at a certain moment in time to be an *agent*. This agent's beliefs are the following: temporary maximal sum s, partial sum t, current data from vector $a[j]$ (DATAIN(a)), and temporary indices i and j.

Its goals are: the left index of the maximal sum sub-array x_1, the right index of the maximal sum sub-array x_2, and the temporary (updated) sums s and t.

The agent's plan is the actual algorithm (Fig. 2):

```
Inputs = (s, t, a[j], i, j)
ag_mem[0] ← Input(0)
ag_mem[1] ← Input(1)
ag_mem[2] ← Input(2)
ag_mem[3] ← Input(3)
ag_mem[4] ← Input(4)
ag_mem[1] ← ag_mem[1]+ag_mem[2]
if (ag_mem[1] > ag_mem[0]) then
 (ag_mem[5], ag_mem[6]) ← (ag_mem[3], ag_mem[4])
  ag_mem[0] ← ag_mem[1]
end
if (ag_mem[1] < 0) then
 ag_mem[1] ← 0
 ag_mem[3] ← ag_mem[1]+1
end
Outputs = (ag_mem[0], ag_mem[1], ag_mem[5], ag_mem[6]) // (s, t, x1, x2)
```

Fig. 2. The agents-based algorithm implementation

When passing from software to hardware, there are a few additional elements to take into consideration.

As indicated in Section 1, the numbers from the input stream are signed integers. It is thus necessary to design the whole architecture in two's complement arithmetic.

A common sense estimation (which also covers a wide range of practical situations) is to consider the input numbers as bytes. Therefore, we need an 8-bit representation of these numbers, and one extra bit for the sign. This is why the DATAIN port is sized as a 9-bit input.

In the 1D case, our purpose was to design a system that works properly for 65,536 numbers in the input stream. Since each number is expressed on 8 bits, a simple estimation shows that, for the worst case situation, we need an *Accumulator* on 25 bits to compute the partial sums (t, in the algorithm presented in Fig. 1). Also, the temporary maximal sum, s, must also be sized as a 24-bit data register, because s will never be negative, as one can easily deduce from the algorithm, so the sign bit is no longer necessary.

Of course, an *Adder* resource is necessary. It does not appear explicitly in Fig. 1, because Kadane's algorithm was targeted for a software implementation. This *Adder* must also be sized on 24 bits.

As can be seen in Fig. 1, the j variable iterates from 1 to n (n being the size of the input stream). In the hardware implementation, this corresponds to a *Counter*. Since our goal was to design a system that deals with streams of at most 65,536 numbers, this *Counter* must be sized on 16 bits.

The algorithm described in Fig. 1 also contains three more variables: x_1, the pointer to the beginning of the maximal subsequence found in the input stream; x_2, the pointer to the end of the maximal subsequence found in the input stream, and i, an intermediate variable that points to the candidate maximal subsequence. These variables are stored into data registers, which must be sized on 16 bits.

When moving the algorithm from software to hardware, some additional resources are necessary, which do not appear explicitly in the pseudo code: the Adder and the Counter are examples of such "hidden" resources. Another hidden resource is the Increment ("+ 2") Block, which is necessary for implementing the (apparently) simple operation $i \leftarrow j + 1$ in the last line (line 11) of the algorithm.

The goal of the hardware implementation is to obtain a speed increase. For this purpose, our intention was to create a pipelined design, where a new number from the input stream is processed in each clock cycle. The whole design must of course be synchronous.

It is generally a good idea to buffer the input data, so, as can be seen in Fig. 3, a new partial sum ($t \leftarrow t + a[j]$) will be obtained one clock cycle later than the arrival of a new input number.

This organization also has another advantage. At the end of the algorithm (lines 10-12 on Fig. 1), we have to compare t to 0. Because of this structure (see Fig. 3), we know in advance if the *Accumulator* (that stores t) will be loaded with a negative number. In software this operation would take place exactly as described by the algorithm: first, load t with the new value ($t \leftarrow t + a[j]$), then test if it is negative. In hardware, since t is a data register loaded with the result produced by the *Adder*, if the first bit is '1' (meaning that the result of the addition is negative), on the next clock cycle we can directly and synchronously reset the *Accumulator*.

Because of this structure, the algorithm must be adapted as follows. At the end of the current iteration, the i variable must be updated to the value $j + 1$. In our case, we must perform the operation: $i \leftarrow j + 2$, because t "will be negative" only later, after one clock cycle. But after that clock cycle, j will be incremented, so i must be updated to the value $j + 2$.

The Comparator is a combinational resource needed to determine if $t > s$ (line 7 in Fig. 1). It must be sized on 24 bits, because s is always positive and is compared to t only when t is positive too, so for t the sign bit is no longer necessary.

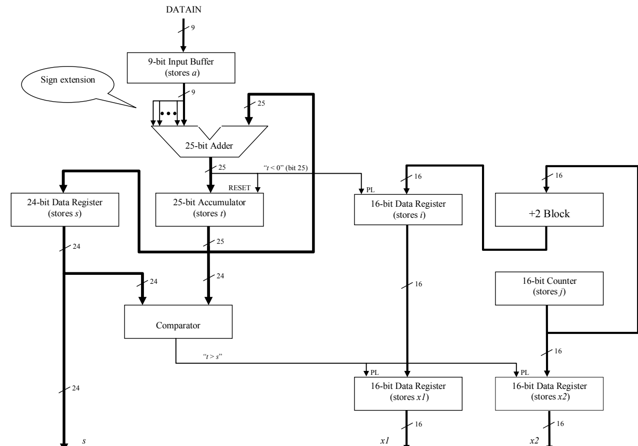

Fig. 3. The hardware architecture in the 1D case

Technically speaking, the sign extension is not a resource, but it is implemented by the interconnection network. We must perform a sign extension on the input numbers, because they come as two's complement 9-bit words, while the Adder is sized on 25 bits.

To preserve the readability of the schematic, the *Clock* and *Preset* signals from Fig. 3 have not been represented, for all the sequential components (the 9-bit Input Buffer, the 25-bit Accumulator, the 24-bit Data Register, the 16-bit Data Registers, and the Counter).

The initialization of the designed system is done as follows. All the sequential components are initialized with 0, except for the 16-bit Counter that stores j, which will be initialized with 2^{16}-2, i.e. "1111111111111110". The reason is given by the particular features of this architecture, as explained above.

3 Results of the 1D Implementation

The hardware design was specified in parameterizable VHDL code and the physical support of the implementation was a Xilinx Virtex II 1000 FPGA, speed grade -6 device.

After running Xilinx Synthesis software, the reports indicated a maximal working frequency of 133 MHz (clock period: 7.5 ns). The simulation was done in ModelSIM and the proper functioning of the design was confirmed.

It is obvious that the performance depends on the length of the input stream. A simple computation/calculation indicates that for an input stream of 65,536 numbers, the total execution time is 0.49 ms, which indicates at least two orders of magnitude improvement over the software implementation (see Table 2).

An examination of the results yielded by the software implementation shows that the run time does not increase linearly with the length of the input stream,

but in case of the hardware implementation, this dependency is linear.

As reported by Xilinx synthesis tools, the FPGA device is used only at 20% of its total capacity, which allows the implementation of the hardware version for much larger input streams.

As a result, the more we increase the length of the input stream, the better the hardware implementation performs when compared with the software one.

It is easy to adapt the hardware architecture to larger sizes of the input stream by simply resizing the resources on a larger number of bits.

4 The 2D Implementation

The 2D case consists of finding the sub-array of maximal sum inside an M x N array. For example, for the array in Fig. 4, the solution is highlighted in gray.

2	-4	5	-5	-9	3	-4
-5	-6	-9	2	-5	4	1
-3	7	-7	5	6	-6	0
-3	5	-8	8	-2	2	-7
-4	1	-1	0	-1	-4	1

Fig. 4. Solution example for the 2D problem

The software solution to this problem is classical and straightforward. The results are shown in Table 2.

Table 2. Results of the software implementation

Input Array Size ($M = N$)	Run time in C++ (sec)
4	0.00062
16	0.00282
64	0.01560
128	0.05160
170	0.16000
512	1.70300
648	3.39100
1024	13.1870
2048	99.2180

The 2D Kadane algorithm is based on the 1D Kadane algorithm. The 1D algorithm is run on the elements of each row of the array (row_1, row_2… row_M), considered as a 1D stream, then, on the sum of each pair of rows ($row_1 + row_2$, $row_1 + row_3$,… $row_{M-1} + row_M$). The solution is given by the maximal sum produced by the 1D Kadane algorithm on these cases. If x_1 and x_2 are the pointers to the beginning and the end of the maximal sub-stream, and R_i and R_j are the two added rows for which the sum is maximal, then the solution is delimited by the rectangle given by the upper-left (R_i, x_1) and the lower-right corners (R_j, x_2).

As one can see in Table 2, the execution time increases exponentially, $O((M \times N)^3)$ with the size of the input array. The hardware implementation starts showing its usefulness for large arrays, where the gain of speed becomes significant.

For the hardware implementation, a series of adaptations of the algorithm had to be done, in order to wisely exploit the logic resources of the Virtex FPGA devices. The input array is stored in a memory. Depending on the space available in the FPGA device, this memory can be implemented inside or outside the chip. Data are read from this memory, word by word, and the hardware architecture is adapted for a pipeline processing style.

In order to extract rows from the memory the way mentioned above, a special Address Generator module had to be designed. It is presented din Fig. 5a.

To minimize the space occupied inside the FPGA chip, the Multiplier from Fig. 5a is a *MULT18x18* Virtex2 primitive. Also, to implement the memory, Virtex BlockRAMs were used. Thus, the speed of the design increases because specialized components are used, and also the bigger capacity of the Virtex FPGA allows us to implement larger designs.

The Address Generator has the task of generating the appropriate addresses for the Memory to ensure the proper functioning of the algorithm. It must first extract Rows 0 to M-1, then Rows 1 to M-1, then Rows 2 to M-1… etc., until Rows M-2 to M-1. Combined with the architecture of the RowBuffer and helped by the Command Unit (Fig. 5b and Fig. 5c), it will help feeding into the Kadane1D module each Row individually, then the sum of each pair of Rows from the input array. This functionality is ensured by the use of the Multiplier for Fig. 5a, and also the two counters.

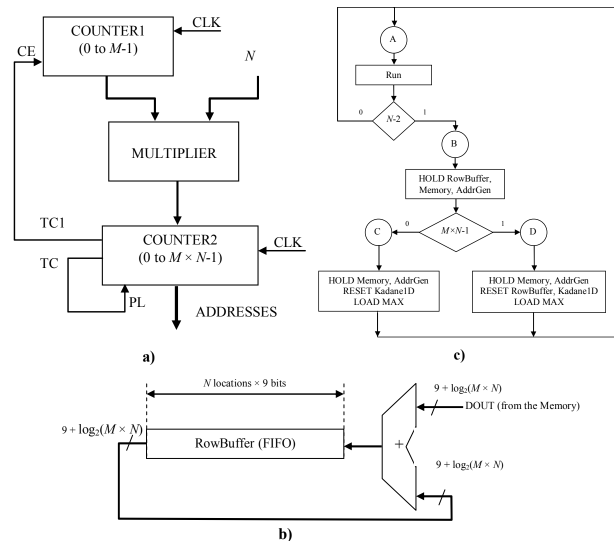

Fig. 5. a) The Address Generator; b) Structure of the RowBuffer; c) State-diagram of the Command Unit

The RowBuffer is a special FIFO module that allows to extract rows from the memory and to add them. Thus, a new word (array element) is extracted in each clock cycle, in a pipeline fashion. The size of this component

is given by the number of columns in the array multiplied by the width of each word (in this case, a nine-bit representation).

An important part of the design is the Kadane1D module, which operates on individual rows or on the result produced by summing pairs of rows, in the order specified above. See Fig. 5b for details.

Here, an adaptation of this module is necessary, modifying the version presented at the beginning of this paper, because by adding the Rows, the size of the words increases. So, the input of the Kadane1D module will be expressed on $9 + \log_2(M \times N)$ bits. This change can also be observed in Fig. 5b.

The output of the Kadane1D module is fed into the MAX module, which will determine the maximal sum, together with some other vital information: the current Row or pair of Rows, the x_1 and x_2 pointers.

The Command Unit controls the functioning of the whole system. It is a relatively simple Finite State Machine (FSM) whose operating mode can be described as in Fig. 5c.

First, a Row (let's call it Row_i) is extracted from the Memory into the RowBuffer. Due to its circular register-like organization, each element of the current Row is added to its corresponding element in the next Row (let's call it Row_j). Simultaneously, the elements of Row_i are fed into the Kadane1D module, so two tasks are performed in parallel:

1) The maximal sub-stream of Row_i is determined in the Kadane1D module;
2) The sum between each element of Row_i and the corresponding element from Row_j is fed into the RowBuffer, ready for the next phase.

The role of the Command Unit is to stop (HOLD) the functioning of the main modules and to synchronously reset those of them who need to be re-initialized.

After a new Row (or the sum of two Rows) is loaded into the RowBuffer, the Command Unit will HOLD the Memory, the Address Generator and the RowBuffer until the Kadane1D module finishes its job of determining the maximal subsequence inside the stream Row_i. As can be seen from Section 2, one more clock cycle is necessary. Then, the result produced by the Kadane1D module is transmitted to the MAX module.

If Row_j is Row_{M-1}, this means the Command Unit also has to reset the RowBuffer (until this moment, we obtained Row_i, $Row_i + Row_{i+1}$, … $Row_i + Row_{M-1}$, but now we have to obtain Row_{i+1}, $Row_{i+1} + Row_{i+2}$, … $Row_{i+1} + Row_{M-1}$, etc.). After that, the processing continues until the last Row is processed.

In this schematic, the control signals are simplified to avoid overcharging and improve the readability of the design. All components have been described in parameterizable VHDL code. Thus, the design becomes portable on any hardware. The only generics to be specified are M – the number of rows, and N – the number of columns of the input array.

The agents will be represented by entities and associated processes, so the inputs Inp (or beliefs) and outputs Outp (or the goals) of the hardware agents will be the ports of these entities. The **AgentName()** pseudo-call simulates "invoking" the agent, or modifying the ports to which its associated process is sensitive.

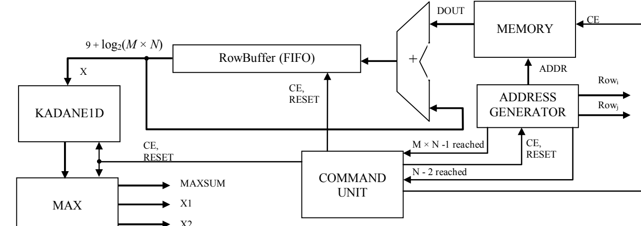

Fig. 6. Kadane2D implementation: general schematic

In the 1D case, we only had one agent, whose goal was to perform the computations in a single loop of the Kadane 1D algorithm. In order to use the **Kadane1D** agent as a part of the Kadane2D-related computations, we need to add a *while* loop:

```
Kadane1D
Inputs = (n, a[1], a[2], ..., a[n])
ag_mem[0] ← 0;  ag_mem[1] ← 0 //x1 and, respectively, x2
ag_mem[2] ← -30000 //s
ag_mem[3] ← 0 //t
ag_mem[4] ← 1; ag_mem[5] ← 1 //i and, respectively, j
ag_mem[6] ← Inputs(0) // n
while (ag_mem[5]<ag_mem[6]) do
    Kadane1DLoop.Inputs(ag_mem[2],ag_mem[3],
Input[ag_mem[5]], ag_mem[4], ag_mem[5])
    Kadane1DLoop()// "invoke" the Kadane1DLoop agent
    ag_mem[2] ← Kadane1DLoop.Outputs(0)
    ag_mem[3] ← Kadane1DLoop.Outputs(1)
    ag_mem[0] ← Kadane1DLoop.Outputs(2)
    ag_mem[1] ← Kadane1DLoop.Outputs(3)
    ag_mem[5] ← ag_mem[5] + 1  // increment
end while
Outputs = (ag_mem[2], ag_mem[0], ag_mem[1]) //s, x1, x2
```

Fig. 7. The updated **Kadane1D** agent

The **Kadane2D** agent is modeled starting from the hardware components it consists of, considering each of them an agent which will be invoked by **Kadane2D**:

MAX: Inputs = (s, x1, x2), Outputs = (maxs, x1, x2, r1, r2)

Kadane1D: Inputs = (a[]), Outputs = (s, x1, x2)

MEM: Inputs = (ADR), Outputs = (DOUT)

AddressGen: Inputs = (N, M), Outputs = (ADR)

Besides several variables, the **Kadane2D** agent also needs to store an array of data at each step. As can be seen in Fig. 8, this array is an actual row from the input matrix, or a sum of such rows; and taking into account the fact that the input matrix has N lines and M columns, the arrays' dimension will be M.

The **Kadane2D** agent's auxiliary storage space (**aux** in the pseudo code) will then be a matrix like this:

	0	1	2	3	4	5	6	7	8	…	M-1
0	N	M	ADR	S	X1	X2	R1	R2	I		
1											
2											

Row0 will store variables; row 1 will store the DOUT array, whereas row 2 will store the ROWBUF array. Thus, the new **Kadane2D** agent's plan will be:

```
Kadane2D
Inputs=(N, M, a[][])
aux[0,0] ← Inputs(0)
aux[0,1] ← Inputs(1)
AddressGen.Inputs(aux[0,0], aux[0,1])
AddressGen()// "Invoke" the address generator module
// And get the address:
aux[0,2] ← AddressGen.Outputs(0)  // ADR
MEM.Inputs(aux[0,2]) // "Invoke" the memory module
MEM()
aux[0,8] ← 0 // Get DOUT!
while (aux[0,8] < aux[0,1]) do // i<m
      aux[1,aux[0,8]] ← Mem.Outputs(aux[0,8])
      aux[0,8] ← aux[0,8]+1
end while
aux[0,8] ← 0 // Get ROWBUF!
while (aux[0,8] < aux[0,1]) do // i<m
 // ROWBUF ← DOUT + ROWBUF
      aux[2,aux[0,8]] ← aux[1,aux[0,8]]+aux[2,aux[0,8]]
aux[0,8] ← aux[0,8]+1
end while
// Kadane1D's inputs are the values in the ROWBUF array:
aux[0,8] ← 0
while (aux[0,8] < aux[0,1]) do  // i<m
      KADANE1D.Inputs(aux[0,8]) ← aux[2,aux[0,8])
      aux[0,8] ← aux[0,8]+1
end while
KADANE1D()// "Invoke" Kadane1D
aux[0,3] ← KADANE1D.Outputs(0) //s
aux[0,4] ← KADANE1D.Outputs(1) //x1
aux[0,5] ← KADANE1D.Outputs(2) //x2
MAX.Inputs(aux[0,3], aux[0,4], aux[0,5])
MAX()// "Invoke" MAX
aux[0,3] ← MAX.Outputs(0)  //maxs
aux[0,4] ← MAX.Outputs(1)  // x1
aux[0,5] ← MAX.Outputs(2)  // x2
aux[0,6] ← MAX.Outputs(3)  // r1
aux[0,7] ← MAX.Outputs(4)  // r2
Outputs = (aux[0,3], aux[0,4], aux[0,5], aux[0,6], aux[0,7])
```

Fig. 8. The Kadane2D agent

5 Results of the 2D Implementation and Conclusions*

The main goal of this paper was to create a hardware implementation of a well-known software algorithm, i.e. Kadane's algorithm, in both its 1D and 2D versions, for determining the maximal subsequence of a stream or an array of integers.

* This work was supported by the Romanian Ministry of Education and Research, under grant AT 178 / 2006.

The paper presents a detailed step-by-step methodology for these adaptations. All choices made for the nature and the size of the used resources are discussed and justified.

Like for the 1D problem, the design was specified in parameterizable VHDL code and implemented in three Xilinx FPGA devices. In fact, the 2D problem implementation is an extension of the 1D design, based on the same working principle. The working frequency is now lower than in the 1D implementation: around 90 MHz, in average, varying with the FPGA family (91.7 MHz for Virtex2, 95.8 MHz for Virtex2PRO and 85.5 for a Spartan3 device). The global performance also depends on the input array's size.

The simulation was done in ModelSIM and the proper functioning of the design was confirmed.

In conclusion, the performance is *at least* two orders of magnitude better in hardware than in software. The space available in Virtex devices allows implementations for large input arrays, the main limitation being introduced by the amount of available on-chip memory. A larger off-chip memory can be used, but lowering the working frequency.

Future work will involve creating a hardware algorithm for determining not only the maximal (or minimal) subsequence, but the first k maximal (or minimal) subsequences in an input stream that represents a 1D or a 2D array.

References:

[1] Bentley, J., "Programming pearls: algorithm design techniques." *Commun. ACM 27*,9 (1984), p. 865.

[2] Bentley, J., "Programming pearls: perspective on performance." *Commun. ACM 27*,11 (1984), p. 1087.

[3] Sung Eun Bae and Tadao Takaoka, "Algorithms for the Problem of *K* Maximum Sums and a VLSI Algorithm for the *K* Maximum Subarrays Problem." *Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'04)*, 1087-4089/04, 2004.

[4] Hamid Reza Naji and B. Earl Wells, "On Incorporating Multi Agents in Combined Hardware/Software based Reconfigurable Systems -- A General Architectural Framework". *Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)*, 2002.

[5] M. Wooldridge, "Agent-based Software Engineering," *In IEE Proceedings on Software Engineering*, 144(1), pages 26--37, February 1997.

[6] X. Meng, V. Chaudhary, Bio-sequence analysis with cradle's 3SoC™ software scalable system on chip, *Proceedings of the 2004 ACM symposium on Applied computing.*