Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

A Design of Transputer Core and its implementation in an FPGA

Makoto Tanaka, Naoya Fukuchi, Yutaka Ooki and Chikara Fukunaga
Tokyo metropolitn University 1-1 Minami-Osawa, Hachoiji, Tokyo, 192-0397, Japan

Abstract. We have made an IP (Intellectual Property) core for Transputer T425. The same machine instructions as Transputer are executable in this IP core (We call it TPCORE). Thus we can run an occam program, which is compiled with occ of occam toolset developed by Inmos. Although TPCORE uses the same register set with the same functionality as Transputer and follows the same mechanisms for the link communication between two processes and interrupt handling, the implementation must be very different from original Transputer. We have extensively used the micro-code ROM to describe any states that TPCORE must take. Using this micro code ROM for the state transition description, we could implement TPCORE economically on FPGA space and achieve efficient performance.

1 Introduction

The programming language occam has been used to construct a parallel processing system with Transputer(s) and their specific network. The Transputer architecture was designed to optimise the execution a program written in occam. Transputer and occam were once used to implement many parallel or distributed systems extensively all over the world in 1980's. However as Inmos ltd. of the day could not supply a new generation Transputer in early 1990's in timely manner, many users gave up to keep Transputer as a core processor of their parallel systems or even they were forced to look for another architectures rather than parallel one for their applications.

There may be still many people like the authors themselves who hope to run occam codes developed for Transputers or design a parallel system with occam. Although the occam compiler has been evolved and facilitated to execute on many platforms (KRoC , the Kent Retargettable occam Compiler project [1]), we could not find easily a hardware object optimised for occam execution like Transputer even technology of hardware implementation on silicon has been significantly evoluted.

In order to develop such a processor like an occam machine, we have firstly analysed the instruction set of Transputer T425 and tried to resolve every instruction in detail, which has been not described explicitly in the data sheet [2]. We then made an IP core (we call it TPCORE) using Verilog/VHDL to able to process all the instruction sets of T425. We have aimed to construct an IP core which can run an occam program compiled, linked, loaded and downloaded with Transputer toolset developed by Inmos [3]. We then made realisation of TPCORE using FPGA, and made some performance test. We report in this article TPCORE development, logical structures we have taken, hardware implementation for the CPU, link, interrupt and process control blocks. Finally we present some results of performance for the execution of occam programs, whcih were compiled with occ.

2 TPCORE fundamental architecture

The overall block diagram of TPCORE is shown in Figure 1. TPCORE comprises a CPU, a Link block, Memory controller and Memory. The memory consists of four 4Kbyte blocks. The Link block has four interfaces (link) to communicate (exchange data) with other TP-COREs.

2.1 Memory controller

The memory controller accepts either the memory access requests of the CPU or of the link interfaces, and modifies the requests to one specified by the memory (device) actually embedded in an FPGA. In this way we can simply modify the verilog code for the memory controller and keep the CPU and link block untouched even if we implement TPCORE in another FPGA which has a different memory device of the size or data transfer technology.

The memory controller manages one address and one data buses. Although the address space can be extended over about 4GB, which is expressed with 32bit, presently we use only 15bit for the address specification (32kB space). In original Transputer, there was a special address space, which we could access it with faster cycle than the other address space. TPCORE handles, however, all the address space uniformly.

We have not made a dedicated communication bus between the CPU and the link. The data exchange between them is performed using the data bus managed by the memory controller. If the link block occupies the memory block for communication with external modules, the CPU is blocked the execution.

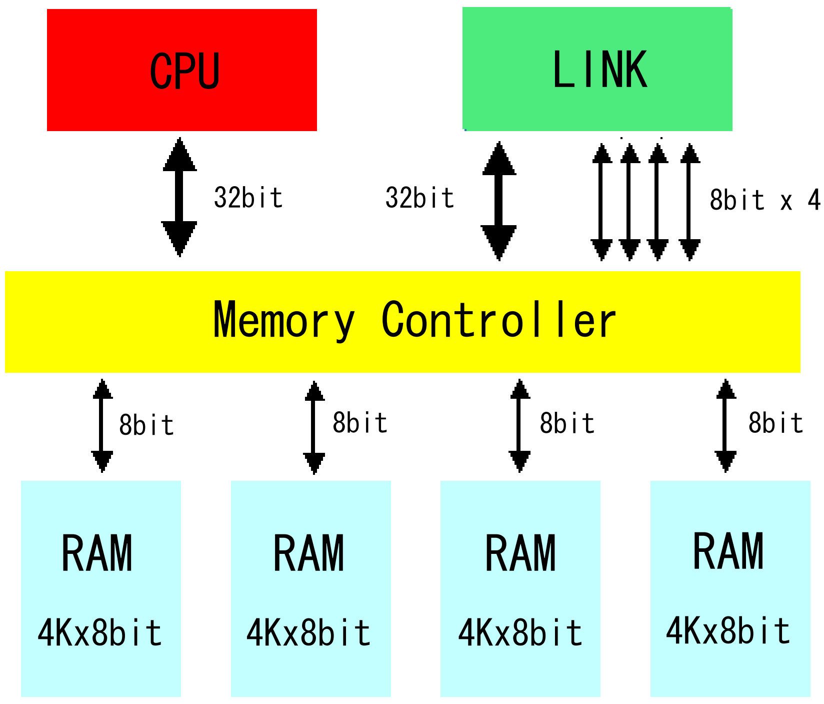


Figure 1: TPCORE block diagram

2.2 CPU

The block diagram of the CPU is shown in Figure 2. The address and data buses shown in the figure are controlled by the memory block. In order to follow the instruction set of Transputer as much as possible, we implement six almost identical registers in TPCORE. These registers are instruction pointer (Iptr), the operand (Oreg), the work space pointer (Wptr), and three stack registers (Areg, Breg and Creg). We have taken the role of these registers as same as the one of Transputer. The value stored in Wptr is recognised as Process ID for a process. The CPU block uses this value to make a local address for the process. The local address for the process is set using Oreg and the lower 4bit of Iptr in addition to Wptr. The least significant bit is used to distinguish the process priority.

Beside these six registers, we have prepared (private) registers for the error handling (Error), loop counting (Cnt), and temporal data storage (Temp). Although the existence of these registers has not been described explicitly in various Transputer data books, it is naturally required for all CPU object. We have implemented them in order not to influence other logic structures reconstructed though the references. Especially we have carefully designed the arithmetic logical unit (ALU) concerning these private registers. ALU has two input and two output streams. As shown in Figure 2, the register Cnt will be a part of the input, and Temp will be one of the output while Error will be a part of both the input and output sources for ALU.

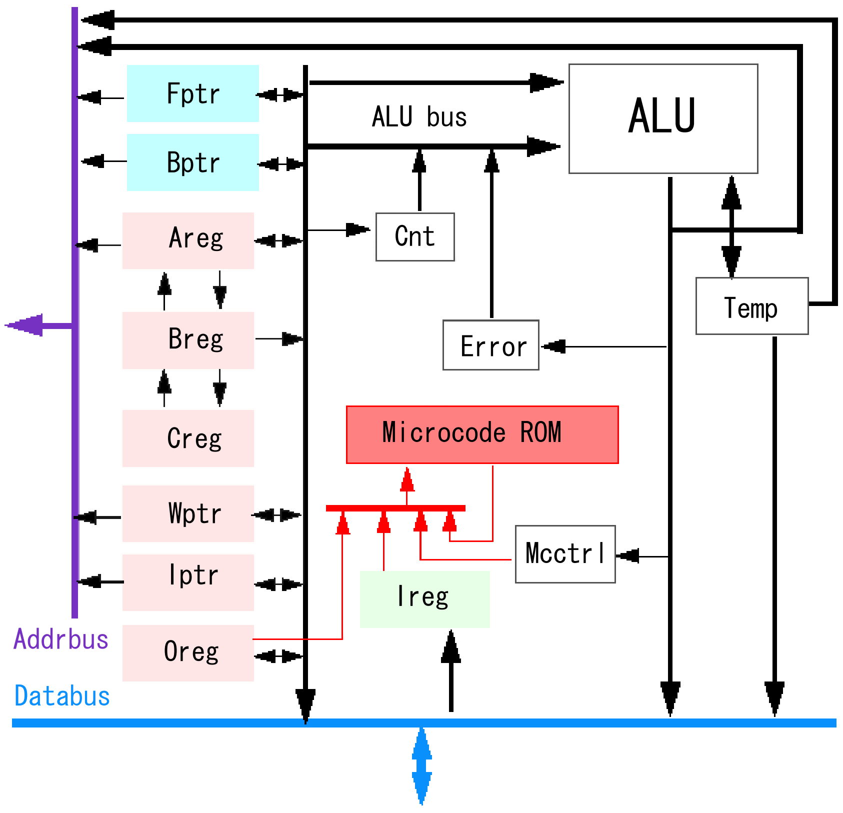


Figure 2: CPU block diagram

Table 1: Micro-code ROM 64bit specification

bit number	Contents
63–61	Not used
60,55,36-33	Condition branch for the micro-code ROM
59–57	memory access privilege
56,54	Behaviour of the Link interfaces
53-50	Output destination for the data bus
49-37,23	Behaviour of various registers
32-24	Input source selection for ALU
22,21	Input selection for the address bus
20-17	Input selection for the data bus
16-14	Process priority
13-9	Instruction code for ALU
8-0	Next address of the micro-code ROM

2.3 State transition table in micro-code ROM

All the possible states in TPCORE for also the memory controller, the link block beside CPU are described in the micro-code ROM. The micro-code ROM has the depth of 512 with the width of 64bit. The contents to describe a state with 64bit are summarised in Table 1.

We have two advantages to use the micro-code ROM for description of the state transition; the one is that we can relatively easily monitor, modify and extend the instruction performance by changing appropriate bits of the appropriate address of the micro-code ROM without modifying the verilog code of TPCORE, and the another one is that we can save the space siginificantly for logic circuits since we pack all the state transitions into an internally embedded memory and are free to put the state machine into the wired-space.

Several examples of the contents of the micro-code ROM are given bellow.

Instruction Fetch State
The address of the micro-code ROM for this state is 0x000. Iptr is set for the source of the address bus where Iptr contains the address for the next instruction. The source for the data bus is set to memory since the address to be executed next is loaded on the address bus. Ireg is set to the destination for the data bus. The next micro-code ROM address is set to 0x00 to go to the instruction decode state.

Instruction Decode State
The contents of four higher bits of Ireg or Oreg 32bit are used to specify the next instruction to be done. The next address of the micro-code ROM is then determined conditionally according to the instruction decoded.

Instruction Execution State
If the instruciton to be executed is finished in one state transition, then the next state will be back to the Instruction Fetch. Instead if the instruction needs another states to complete, then the next address for the micro-code ROM is an appropriate one for the next state.

3 Hardware for the parallel processing

3.1 Process control

The mechanism of the process control in TPCORE follows basically the one used for Transputer as faithfully as possible. The value in Wptr is regarded as the process ID. The first address to be executed in the process is stored in Wptr-4, and the ID of the next process to be executed is stored in Wptr-8. Thus the process itself has the information for the next process. This link structure for the process queue is prepared separately for the high and low priorities, and the structures are retained in the registers of fptr0, fptr1, bptr0 and bptr1. Since a change of one of these register in the process scheduling is regarded as the state transition, the process control is also managed by the micro-code ROM.

3.2 Interrupt process

The mechanism for the interrupt handling is followed from Transputer. The save or reload of the relevant registers at the beginning and end of an interrupt is described in state changes. The handling of the interrupt is described with the micro-code ROM. Once an interrupt is occurred, the address for the next micro-code ROM is changed to point the addresses to initiate or terminate the interrupt handling (18 and 22 states for the interrupt and return from interrupt respectively). Afterwards normal state transition cycle is resumed.

3.3 Link communication

The communication between two processes in TPCORE is done through channels as is done in Transputer. The communication is one to one and synchronous. The channel facilitates no buffer for data to be transferred. A 32bit word in the memory is used for a channel between two processes running in the same TPCORE while one of total four link interfaces (also implemented in a special address space in memory) is used for a channel to communicate with a process running in a different TPCORE. The assembly instructions for communication like `in` and `out` distinguish internal and external communication from the address used for the channel. The stack register Creg is used for a pointer to specify the address of the data, Breg is the channel address, and Areg is used to specify the number of bytes to be transferred.

If, for example, `out` is executed with Breg=0x80000000, the link interface0 will be used to output message externally, and the CPU asks to the link block to do the external communication by giving contents of the stack registers and the current process ID. Suspending the execution of the process currently executed, the CPU starts the next process taken from the scheduling queue. Once the link communication is over, then the CPU restores the stack registers and the process ID, and resumes the suspended process. The link protocol for the external communication is the same one as defined as Inmos protocol. The TPCORE has a link interface to accept the data over RS232C line. The protocol for data transfer over RS232C is the same one as the Inmos link protocol.

4 Implementation and verification

The design of the TPCORE Hardware has been done with the following steps,

1. **Analysis of the Transputer instruction set**
   In order to investigate what changes are occurred in the internal registers or memory for execution of an instruction, we have extensively used a program ”isim”, which is an application involved in Inmos Transputer toolset [3]. As we could observe changes of the registers and relevant memories with Transputer instructions one by one, isim was very useful tool to look into the internal state transitions caused by some complicated instructions such as ones associated with PAR or ALT constructions.

2. **Description of the micro-code ROM**
   Once the changes in the registers or memories by the instructions were understood, we have summarised it in the framework of the state transition model. The model is implemented into the micro-code ROM. We also include state transitions caused by the interrupt, external link communication etc. into the micro-code ROM.

3. **Hardware design**
   Once the format of the micro-code ROM has been established and the contents of it have been filled, we have begun to design the hardware parts (the CPU block, memory controller, and the link block) using verilog. The verilog code was verified with the simulation. The verification of the hardware design also contains the validity check of the description in the micro-code ROM. An example of the simulation is shown in Figure 3. We have used ModelSimXEII5.6e [4] for the verilog simulation of both the register transfer and gate levels, and used ISE6.1i [5] for the logic synthesis.

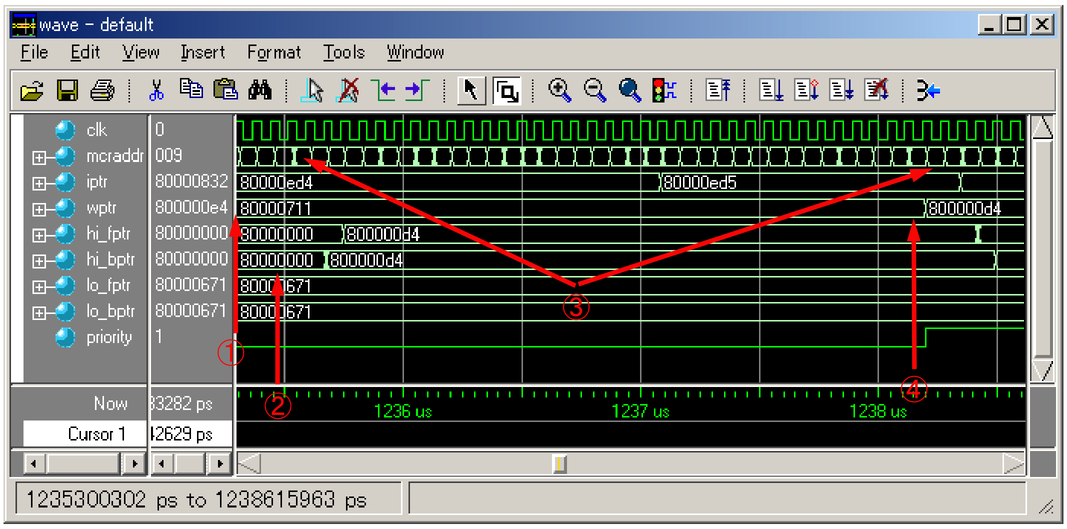


Figure 3: An example of simulation output - generation of an interrupt: The detailed discussion for this figure will be found in the text

The figure shows following steps of the simulation;

1. the lowest bit of Wptr is set to one as TPCORE executes a lower priority process in the beginning,

2. a higher priority process is generated, and its Wptr (0x800000bc) is put in hi_fptr at about 1235.5us,

3. the address of the micro-code ROM (mcraddr) must be changed simply from 0x16e to 0x000 unless an interrupt is generated, but is changed to 0x000 through 0x1d3, 0x1d4,. . . ,0x1e4, these 18 extra addresses of the micro-code ROM contains the states during the interrupt handling, and

4. after the state transition by the interrupt is over at around 1238.5us, a higher priority process is going to be executed.

We have implemented TPCORE developed in this way on an FPGA of Xilinx Virtex II. The result of this implementation is summarised in Table 2.

Table 2: Implemetaion detail

Working frequency	24MHz (max. estimated 31.5MHz)
Number of gates	1371928 (1.4M) gates (64% used)
Memory size	32kByte
Memory access rate	24MByte/s
Number of instructions	96

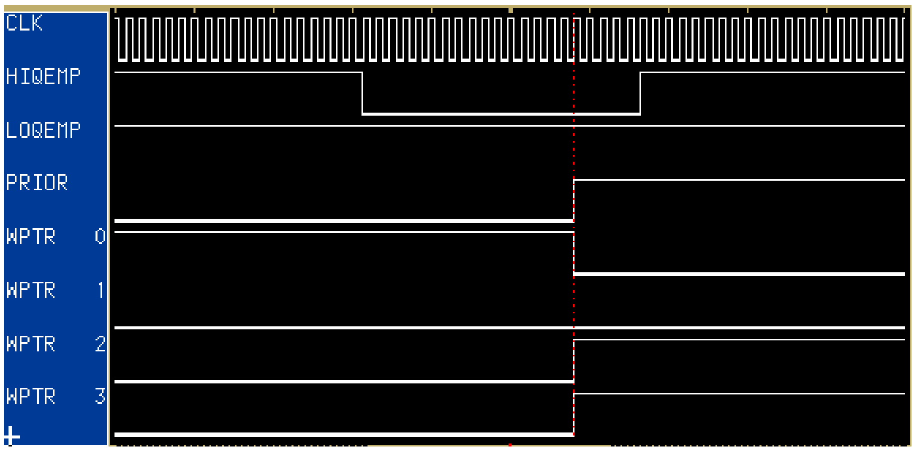


Figure 4: A signal sequence observed inTPCORE implemented in Xilinx Virtex II for interrupt generation. The detailed discussion of this figure will be found in the text

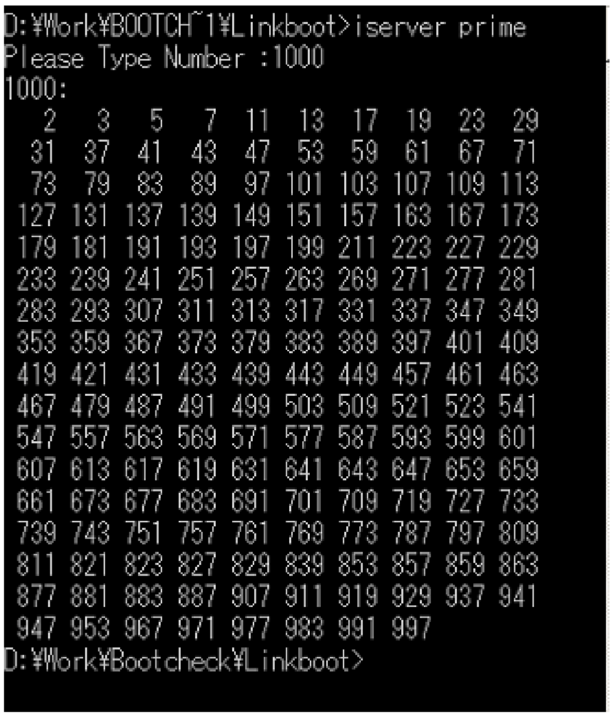
Figure 4 shows a signal sequence actually observed in TPCORE implemented in Virtex II. This sequence expresses an interrupt generation. In Figure 4 HIQEMP and LOQEMP mean empty flags of high and low priority queues respectively, PRIOR indicates the priority of the process currently being excuted. The WPTR0 to WPTR4 are the four least significant bits of Wptr. The sequence of the signals in the figure is interpreted as follows,

1. a low priority process is being executed since Wptr0 is set to high at the beginning,
2. HIQEMP is transited to low at some time, namely a high priority process is entered in a waiting queue, and
3. after some clocks, the lowest significant bit of Wptr is changed from high to low, this indicates the high priority process was entered in an execution state from the waiting queue.

Finally we demonstrate two examples of the occam program execution （one is a prime number search with the algorithm of so called the sieve of Eratosthenes (Figure 5), and the other one is Tour of a Knight on a chess board (Figure 6). The programs were loaded by iserver into TPCORE and the output messages were printed on the host PC screen with various subroutines in hostio.lib of the occam 2 toolset library.

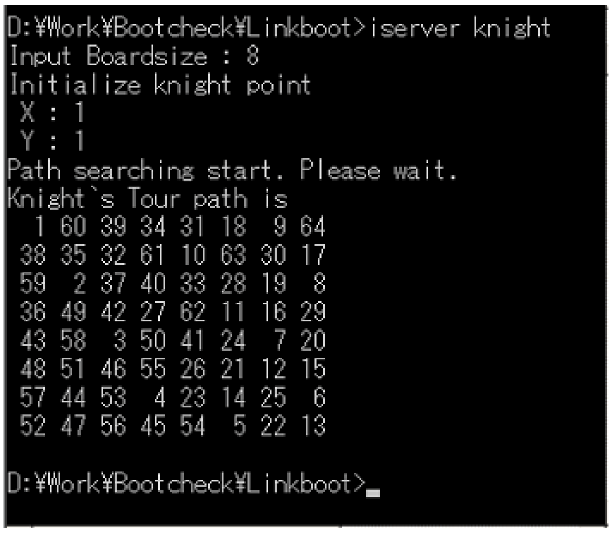
5 Summary and outlook

We have made an IP core of Transputer T425, called TPCORE. TPCORE can execute a program written in occam, which is complied with occ, linked with ilink. We can use iserver



```
D:¥Work¥BOOTCH~1¥Linkboot>iserver prime
Please Type Number :1000
1000:
   2   3   5   7  11  13  17  19  23  29
  31  37  41  43  47  53  59  61  67  71
  73  79  83  89  97 101 103 107 109 113
 127 131 137 139 149 151 157 163 167 173
 179 181 191 193 197 199 211 223 227 229
 233 239 241 251 257 263 269 271 277 281
 283 293 307 311 313 317 331 337 347 349
 353 359 367 373 379 383 389 397 401 409
 419 421 431 433 439 443 449 457 461 463
 467 479 487 491 499 503 509 521 523 541
 547 557 563 569 571 577 587 593 599 601
 607 613 617 619 631 641 643 647 653 659
 661 673 677 683 691 701 709 719 727 733
 739 743 751 757 761 769 773 787 797 809
 811 821 823 827 829 839 853 857 859 863
 877 881 883 887 907 911 919 929 937 941
 947 953 967 971 977 983 991 997
D:¥Work¥Bootcheck¥Linkboot>
```

Figure 5: Prime number search with ”sieve of Eratosthenes” method executed in TPCORE



```
D:¥Work¥Bootcheck¥Linkboot>iserver knight
Input Boardsize : 8
Initialize knight point
 X : 1
 Y : 1
Path searching start. Please wait.
Knight`s Tour path is
  1 60 39 34 31 18  9 64
 38 35 32 61 10 63 30 17
 59  2 37 40 33 28 19  8
 36 49 42 27 62 11 16 29
 43 58  3 50 41 24  7 20
 48 51 46 55 26 21 12 15
 57 44 53  4 23 14 25  6
 52 47 56 45 54  5 22 13

D:¥Work¥Bootcheck¥Linkboot>_
```

Figure 6: Tour of a Knight on a 8   8 chess board

for download of the executable program from a host PC to TPCORE. We expressed all the state transitions caused by execution of the CPU instructions, link and interrupt processing as well as process scheduling, and put them into the micro- code ROM. This implementation makes us easier modification and extension of the TPCORE performance and saves resource in an FPGA.

Almost all the instructions prepared for Transputer T425 have been successfully implemented into TPCORE whereas there are some instructions not implemented in TPCORE. The instructions concerning time sharing have not been implemented actually in TPCORE. These instructions are inserted by the occam compiler automatically, for example, when a long loop instruction is used in an occam program. The detailed behaviour of these instructions are neither given in [6] or got through isim running. Thus we have left unimplemented yet the instructions in TPCORE. In order to implement these time sharing instructions, we must execute these instructions in Transputer actually and debug the relevant registers. This is the issue to be done in the next step.

Although the occam programs demonstrated in the previous section use the constructors PAR or ALT, the parallelization or multiplexing of processes are done within a single TPCORE. We have not yet checked the validity of the link block logic so carefully and hence the communication with a process running in another TPCORE. By upgrading FPGA or increasing the number of FPGA, we will soon start the validity check of the external link communication with this block.

References

[1] Kent University KRoC Web cite: http://www.cs.kent.ac.uk/projects/ofa/kroc/

[2] SGS-THOMSON Micorelectronics, Transputer IMS T425 data sheet 1996

[3] SGS-THOMSON Micorelectronics, IMS D0305 ”Occam 2 Transputer toolset Reference Manual” 1983

[4] Mentor Graphics Corporation, http://www.mentor.com

[5] Xilinx, Inc., http://www.xilinx.com

[6] Inmos Ltd., ”Transputer Instruction set –A Compiler Writer’s Guide–”, Prentice Hall 1988,