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

Design of a Transputer Core and its Implementation in an FPGA

Makoto TANAKA, Naoya FUKUCHI, Yutaka OOKI and Chikara FUKUNAGA
Tokyo Metropolitan University 1-1 Minami-Osawa, Hachoiji, Tokyo, 192-0397, Japan

Abstract. We have made an IP (Intellectual Property) core for the T425 transputer. The same machine instructions as the transputer are executable in this IP core (which we call the TPCORE). To create an IP core for the transputer has two aspects. On one hand, if we could succeed in building our own one and putting it in an FPGA (or VLSI chip), we could apply it as a core processor in a distributed system. On the other hand, if we can extend our transputer development from a very conventional one to more sophisticated ones, as Inmos proceeded to the T9000, we hope to find technological breakthroughs for the bottlenecks that the original transputer had – such as the restriction of the number of communication channels. It is important to have an IP core for the transputer. Although the TPCORE uses the same register set with the same functionality as the 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 micro-code ROM to describe any states that the TPCORE must take. Using this micro-code ROM for the state transition description, we could implement the TPCORE economically on FPGA space and achieve efficient performance.

1 Introduction

The transputer was once widely used as a core processor in parallel or distributed systems extensively all over the world in the 1980s. However as Inmos Ltd. of the day could not supply a new generation transputer in the early 1990s in timely manner, many users gave up using the transputer as a core processor of their parallel systems or were forced to look for architectures other than a 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 to design a parallel system with occam. Although the occam compiler has been evolved and facilitated to execute on a Linux machine (KRoC , the Kent Retargettable occam Compiler project [1]), we could not find easily a hardware object, which is optimized for occam execution like the transputer – even though the technology of hardware implementation on silicon has significantly developed.

We have two motivations to have an IP core of transputer: one is an intention still to apply the transputer in our home-made distributed systems; and the other one is an intention to make a start point to find a solution for new transputer architecture rather than T9000. If we have the IP core, we may be able to try to find our way to overcome, for example, the case of an excess of the number of communication channels over the number of physical links between processes running in different transputers; we may propose new schemes for load distribution and scheduling algorithms; and we may find an idea for a fast cross-bar switching algorithm by implementing multiple cores into one chip.

In order to develop such a processor like an occam machine, we have firstly analyzed 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 (TPCORE), using Verilog/VHDL, to be able to process all the instruction set of T425. We have aimed to construct an IP core which can run an occam program compiled, linked, loaded and downloaded with the transputer toolset developed by Inmos [3]. We then made a realization of TPCORE using FPGA, and carried out some performance tests. We report in this article TPCORE development, logical structures we have chosen, hardware implementation for the CPU, link, interrupt and process control blocks. Finally, we present some results of performance for the execution of occam programs, which were compiled with occ.

2 Fundamental Architecture of TPCORE

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 TPCOREs.

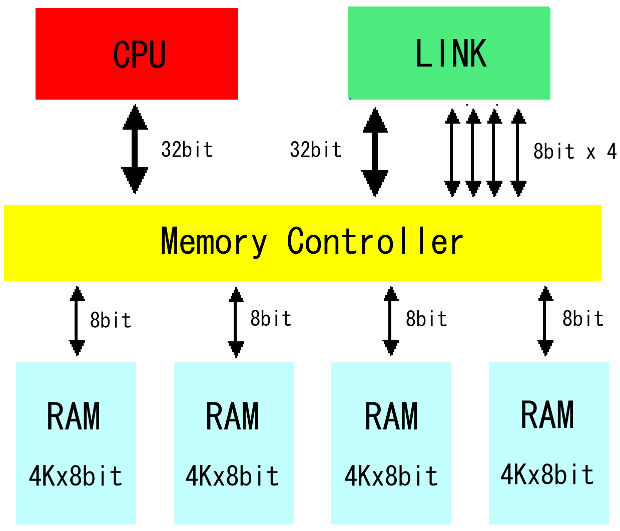


Figure 1: TPCORE block diagram

2.1 Memory Controller

The memory controller accepts either the memory access requests from the CPU or from the link interfaces, and adjusts the requests according to the specification of the 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 bus. Although the address space can be extended over about 4GB, which is expressed with 32 bits, presently we use only 15 bits for the address specification (32kB space). In the 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 common data bus managed by the memory controller. If the link block occupies the memory block for communication with external modules, execution in the CPU is blocked.

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 the transputer as much as possible, we implement six almost identical registers in TPCORE. These registers are the instruction pointer (Iptr), the operand (Oreg), the work space pointer (Wptr), and three stack registers (Areg, Breg and Creg). We have given these registers identical roles to the ones of transputer. The value stored in Wptr is recognized 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.

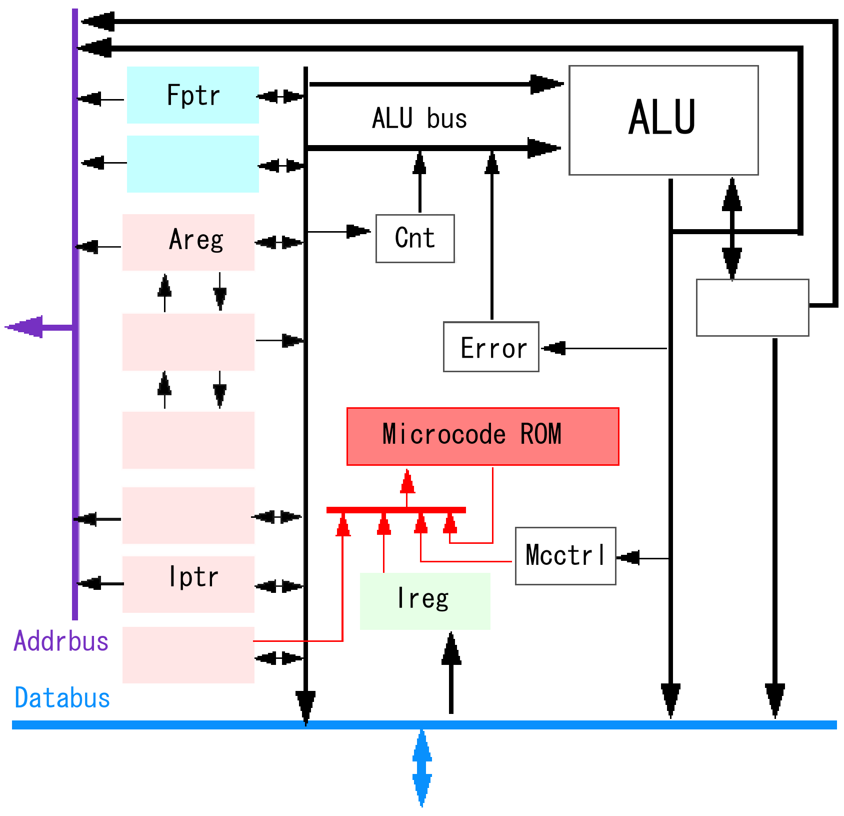


Figure 2: CPU block diagram

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 to be installed in a 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 an input, and Temp will be an output while Error will be either input or output source of ALU.

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	Behavior of the Link interfaces
53-50	Output destination for the data bus
49-37,23	Behavior 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 possible states described with items listed in Table 1 in all the components of TPCORE (the memory controller, the link block, and the CPU) are stored in the micro-code ROM. The micro-code ROM has the depth of 512 and the width of 64bit.

We have found two advantages to use the micro-code ROM for description of states and their transitions; one is that we can modify and adjust performance of an instruction 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 reduce the FPGA space since we pack all the state transitions into an internally embedded memory, and we do not need to install state transition machines into the FPGA wired-space.

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

Instruction Fetch State
In order to fetch the instruction to be executed next:

1. Iptr must be selected to Input for the address bus in which Iptr contains the address for the next instruction,
2. memory must be selected to the source for the data bus since the address to be executed next which is kept in Iptr must loaded on the address bus,
3. Ireg must be set to the output destination for the data bus, and
4. the next address of the micro-code ROM must be set to 0x001 to go to the instruction decode state.

The specification is given in this state and is described in the micro-code ROM at address 0x000..

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 instruction 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 other states to complete, then the next address for the micro-code ROM is an appropriate one for the next state.

3 Hardware for 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 chain 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 registers 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 also derived from the one used in 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 the 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 a total of 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 the 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.

3.4 ALT Procedure

TPCORE implements `ALT` construction in the following way.

1. Address (Wptr-12) is prepared for `ALT` processing to keep the status of the `ALT` process. There are three statuses; Enabling, Waiting and Ready.

2. The status becomes Enabling when `alt` instruction is executed at the beginning of `ALT` construction.

3. Then `enbc` is executed to check whether a guard channel already received data. If so, then the status is set Ready, and `altwt` is over. If not, `altwt` sets the status to Waiting, and yields other process to proceed.

4. An `out` instruction of an another process, which is linked with an input guard channel of the `ALT` recognizes that `altwt` is being executed, and set remotely Ready to the status, and `altwt` is terminated.

5. If a guard channel receives data, an interrupt is generated to resume the `ALT` process, `disc` instruction is executed to sweep out `altwt` remnant, and it determines an appropriate address for execute of instructions for the established guard, then `altend` is executed to jump the address that `disc` specified.

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 summarized 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 shows following steps of the simulation:

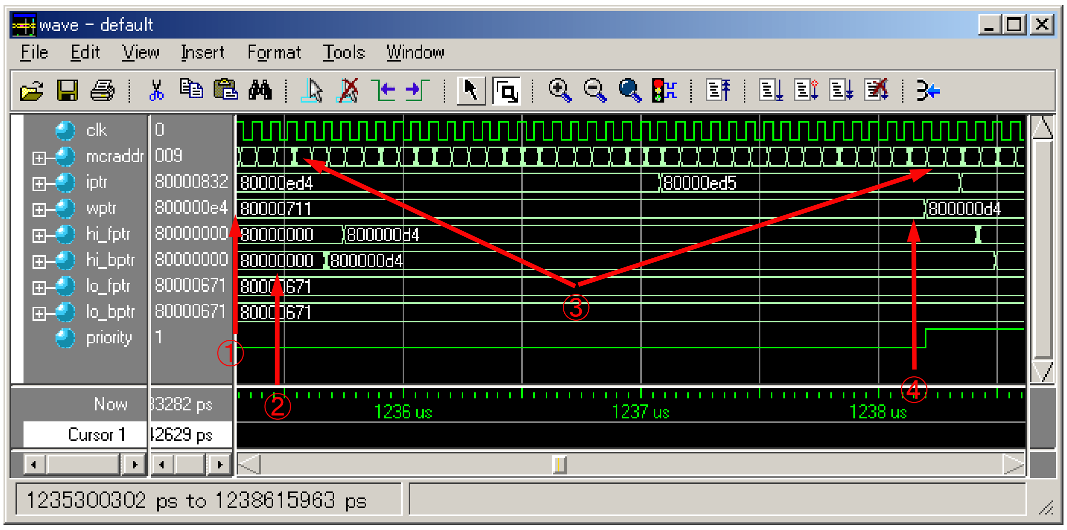


Figure 3: An example of simulation output: generation of an interrupt.

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 (0x80000711) 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.

Table 2: Implementation 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

We have implemented TPCORE developed in this way on an FPGA of Xilinx Virtex II. The result of this implementation is summarized in Table 2. Note that TPCORE has implemented only 96 instructions while the T425 has 103. We have implemented no timer instructions as yet.

TPCORE must do the following steps (some routes are indicated in Figure 2) within one cycle of the clock in order to do (a part of) an instruction;

1. input sources for ALU are assigned to ALU buses by micro-code ROM controller (Mc-ctrl) according to the micro-code ROM description,
2. ALU put an output as a result on the ALU output bus and sets various condition codes, and

3. Mcctrl decodes the data on the ALU output bus and calculates the next micro-code ROM address to refer.

TPCORE will do one 64bit subtraction, one 32bit addition and three steps of 32 to 64bit multiplexing operations for the above process in one clock. This complication limits the working frequency to 24MHz in the FPGA implementation.

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

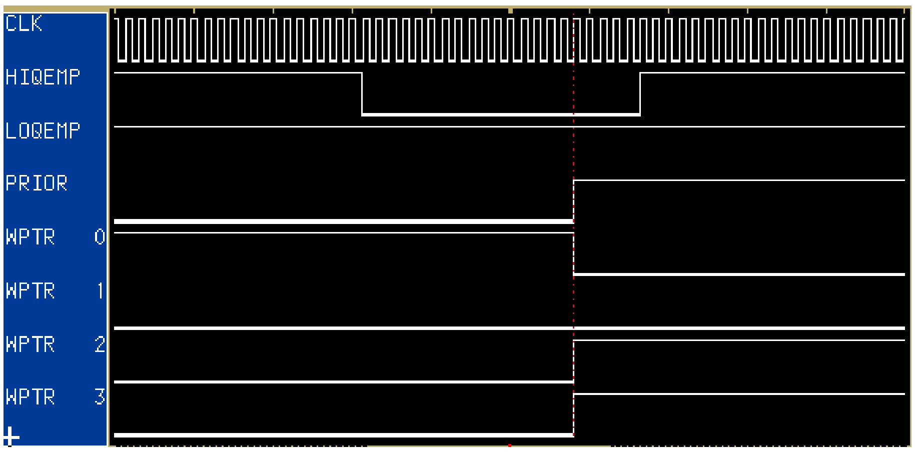


Figure 4: A signal sequence observed in TPCORE implemented in Xilinx Virtex II for interrupt generation.

mean empty flags of high and low priority queues respectively, PRIOR indicates the priority of the process currently being executed. 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.

In Table 3 we listed comparison of TPCORE with T425 for the number of cycles needed for typical instructions. PS and B used in Table 3 denote the number of cycles needed to change processes and the highest bit number in which 1 is set in Areg, respectively. In TPCORE the number of cycles needed to change an active process depends on the conditions (priorities and number of queues). An interrupt is occurred when a process is put in a high priority queue during execution of a low priority process. In this case we need 18 cycles to exchange processes. It takes only four cycles to exchange two low priority processes. The column for the number of cycles for altwt (alt wait) in T425 in this table has been left blank. Accroding to [6], number of cycles for this instruction in T425 can not be explicitly defined since timer instructions will be used for the guard. Timer instructions have not been implemented in TPCORE, the number for TPCORE for altwt is one in case of not using timer instructions.

Finally we demonstrate two examples of the occam program execution; one is a prime number search with the algorithm of the so-called ‘Sieve of Eratosthenes’ (Figure 5), and the other one is ‘Knight’s Tour’ on a chess board (Figure 7). The programs were loaded

Table 3: Comparison of the number of cycles needed for typical instructions for TPCORE with T425.

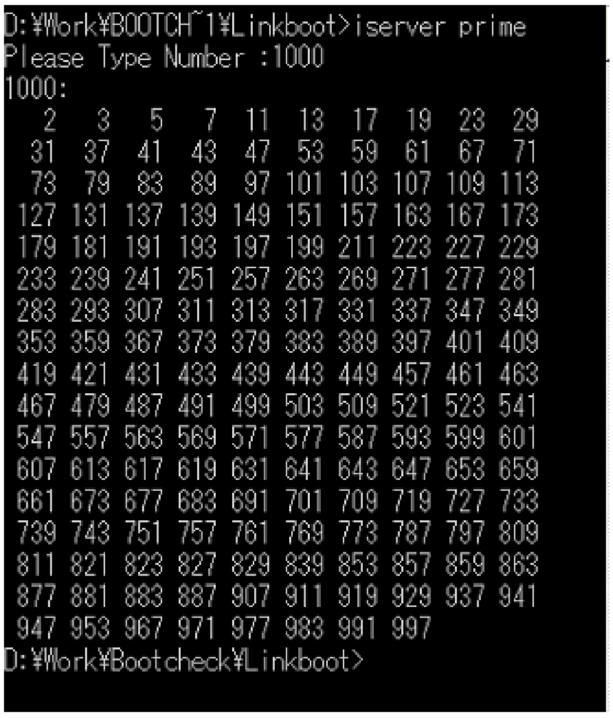
Instruction	Description	T425	TPCORE
j	jump	3	1
ldc	load constant	1	1
ldl	load local	2	2
ldnl	load non local	2	2
eqc	equal constant	2	1
pfix	prefix	1	1
call	call	7	7
wcnt	word count	5	6
in (internal)	input mssg	16	16+7B+PS
out (internal)	output mssg	16	16+7B+PS
altwt	alt wait		7+PS
enbc	Enable Channel	7	8
add	add	1	1
rem	remainder	37	45

by iserver into TPCORE and the output messages were printed on the host PC screen with various subroutines in `hostio.lib` of the occam2 toolset library.

Figure 6(a) shows the elapsed time of the prime number search program (the Sieve of Eratosthenes) versus the integer upper limit for the search range. Since we have not installed any instructions related to timer, we have measured the time with a logic analyzer. An occam code fragment for testing the primeness of an integer (denoted as `max` in the code below) is shown below.

```
SEQ
  j, check, going := 2, TRUE, TRUE
  WHILE going
    SEQ
      pcheck := max REM j
      IF
        max = j
          check, going := TRUE, FALSE
        pcheck = 0
          check, going := FALSE, FALSE
        TRUE
          j := j+1
```

The elapsed time is not linear with the search region. If we count, however, the number of repeat times for this loop in a search and plot the execution time versus this count, we could find a linear relation between two quantities as shown in Figure 6(b). One can calculate 4.8 microseconds/loop from the slope of the line in this figure. The number of cycles needed to execute this loop once is expected as 74 after we analyze the assembler code for this part. We find, therefore, that one cycle needs about 66ns, which corresponds to 16MHz working frequency. We are still analyzing reasons that this frequency is far below 24MHz; the norminal working frequency of TPCORE.



```
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: ‘Sieve of Eratosthenes’ executed in TPCORE for prime number search.

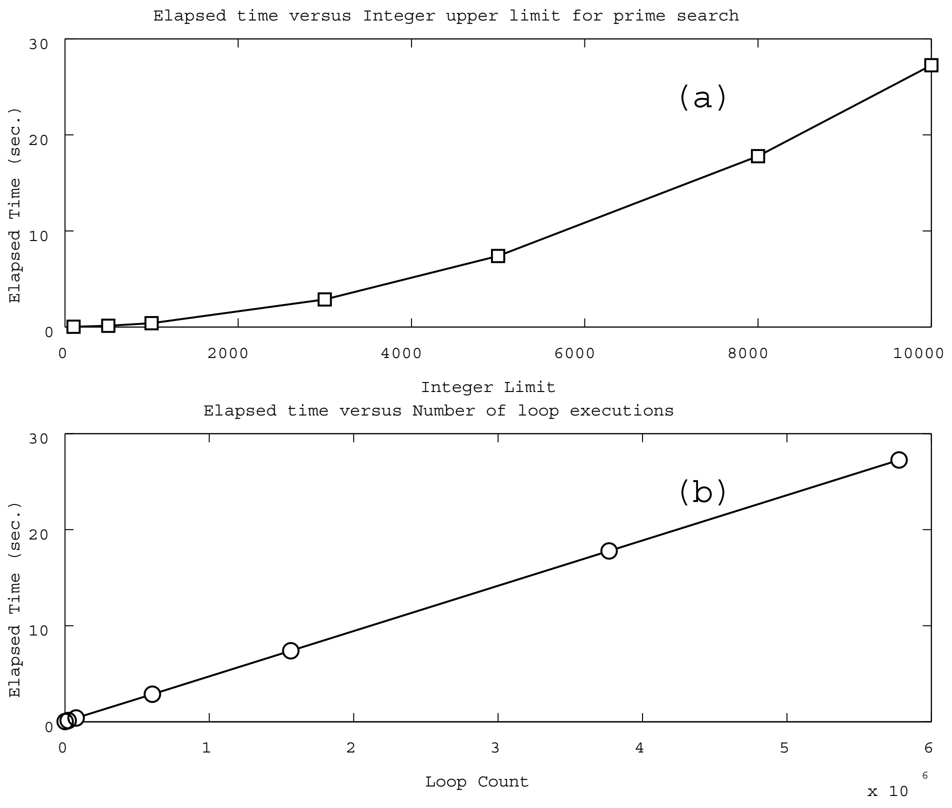
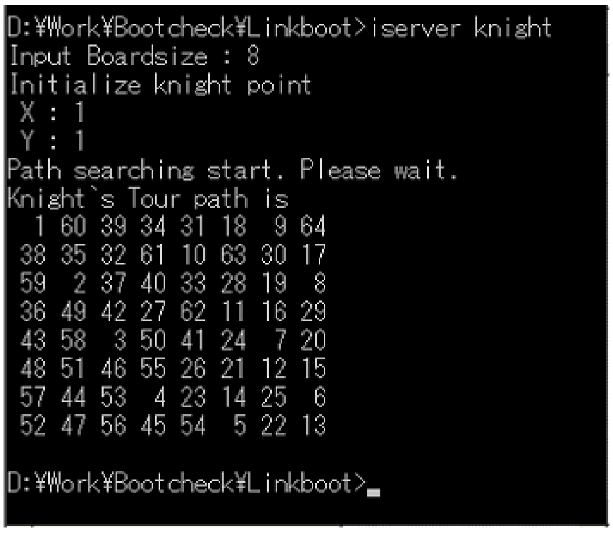


Figure 6: Performance of prime number search with ‘Sieve of Eratosthenes’ method executed in TPCORE.



```
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 7: ‘Knight’s Tour’ on a 8 by 8 chess board.

5 Summary and Outlook

We have made an IP core of the transputer T425, called TPCORE. TPCORE can execute a program written in occam, which is compiled with `occ` and linked with `ilink`. We can use `iserver` to download 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 allows easier modification and extension of TPCORE performance and saves resource in an FPGA.

Almost all the instructions prepared for transputer T425 have been successfully implemented into TPCORE. Instructions concerning time sharing have not yet been implemented 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 obtained through `isim` running. To implement these time sharing instructions, we must execute them in an actual transputer and debug the relevant registers. This is an 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 particularly carefully and, hence, communications with a process running in another TPCORE. By upgrading the FPGA or increasing the number of FPGAs, we will soon start the validity check of external link communications with this block.

Acknowledgement

We acknowledge Dr. M. Imori of the University of Tokyo for his advice and comments during the study. We are grateful to Mr. K. Matsui of Prominent Network Inc. for giving us the information about the CPA conference and suggesting us to submit a paper based on this study to this conference.

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,