Communicating Process Architectures 2015
K. Chalmers, M.J. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2015

OpenTransputer: Reinventing a Parallel Machine from the Past

Andrés AMAYA GARCÍA [1], David KELLER and David MAY

Department of Computer Science, University of Bristol, UK

Abstract. The OpenTransputer is a new implementation of the Transputer first launched by Inmos in 1985. It supports the same instruction set, but uses a different microarchitecture that takes advantage of today's manufacturing technology; this results in a reduced cycle count for many of the instructions. Our new Transputer includes support for channels that connect to input-output ports, enabling direct connection to external devices such as sensors, actuators and standard communication interfaces. We have also generalised the channel communications with the support of virtual channels and the design of a message routing component based on a Beneš switch to enable the construction of networks with scalable throughput and low latency. We aim to make the OpenTransputer and switch components available as open-source designs.

Keywords. parallel architecture, processor, Transputer, Beneš network

Introduction

Over the last few decades, hardware engineers have optimised their processors to achieve higher clock rates, while faithfully increasing the transistor count in their designs as predicted by Gordon Moore in 1975 [1]. Nowadays, it is difficult to achieve higher clock rates due to physical constraints within processors such as heat dissipation and power consumption. Furthermore, processors have grown extremely complicated and difficult to develop. Perhaps more important, these devices are not easy-to-use for software engineers and do not provide enough support to construct large systems where concurrency is inherent. During the 1970s a team of engineers from Inmos recognised this problem and designed a processor known as the Transputer [2,3].

The Transputer is a device intended for parallel computing, it included processor, memory and external links on the same chip essentially making it a computer on a chip. This was supposed to allow information systems to be designed at a higher level – the Transputer functioning as a building block for parallel computing networks.

The Transputer was first presented by Inmos in 1984 during a conference in Tokyo and it was used in many applications including set-top boxes, satellites [4], supercomputers [5] [6] and even a spacecraft [7]. Unfortunately, the Transputer was released in a time when parallelism was not the main concern, so it did not receive the attention it deserved. However, over the last few years, with the shift to cloud computing there has been a trend in the technology world to build large clusters of powerful computers that serve data to an ever-growing number of client devices, which themselves only feature tiny and low-powered processors. These currently include mobile phones and tablets, but will soon also comprise every other

[1]Corresponding Author: *Andrés Amaya García, Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, UK*. E-mail: aa1399@bristol.ac.uk.

device that connects to the internet, ranging from washing machines to cars. We think that the Transputer and its unique feature set make it an excellent processor for applications such as the emerging Internet of Things (IoT) where concurrency management and inter-process communication are paramount.

We present the OpenTransputer [8], a re-implementation of the Transputer that maintains the original Instruction Set Architecture (ISA) whilst updating the microarchitecture (Section 2.1). In other words, the OpenTransputer retains all the ideas of the Inmos processor for concurrency management and inter-process communication, yet the implementation takes advantage of state-of-the-art manufacturing technologies and design techniques. Furthermore, we replaced the communication mechanism used in the 1980s Transputer by a network implemented by switches (Section 2.2.1) to improve the usability of the processor as a building block for large parallel systems. Finally, we introduced an easy-to-use I/O interface (Section 2.3) that can be used to connect hardware peripherals, such as sensors, commonly used in IoT applications.

1. Background

A Transputer consists of CPU, memory and four serial communication links. Since the OpenTransputer is an implementation of the Transputer ISA, we provide a brief description of the main features of this architecture and the occam programming language.

1.1. occam

occam is a programming language designed at Inmos to facilitate the development of concurrent, distributed systems [9]. The language was derived from Hoare's work on Communicating Sequential Processes (CSP) [10] and was developed hand-in-hand with the Transputer to extract maximum performance from the architecture. Despite its purpose, occam is a high-level language and not assembly.

occam enables systems to be described as collections of concurrent processes that communicate with each other. Contrary to conventional languages, occam programs are based on *processes* rather than procedures [11]. Procedures are sequences of instructions that are enclosed into callable constructs enabling code reusability and modularity. On the other hand, processes can be thought of as self-contained programs – with their own threads of control, separate memory spaces and state – that also enable code reusability and modularity. These processes are designed to run on system components containing on-chip memory and communication links to other components (e.g. Transputers). The inter-process communication model is based on CSP and message passing [12]. occam exposes point-to-point channel primitives that directly describe the connections between processes. The *compositional* nature of CSP semantics carries over to occam, simplifying the design, construction and maintenance of complex systems.

occam programs can be executed by a single Transputer that schedules the processes on its CPU. Alternatively, the same concurrent programs can be executed by a network of Transputers without changing the occam description. In this case, processes truly run in parallel and communicate by using the links.

1.2. Transputer Architecture

In the Transputer, each process is associated with a workspace in memory [13]. This area can be thought of as a stack where local variables, channel information and other values that describe the state of the process are stored. When a process is executed, its context is held in six registers:

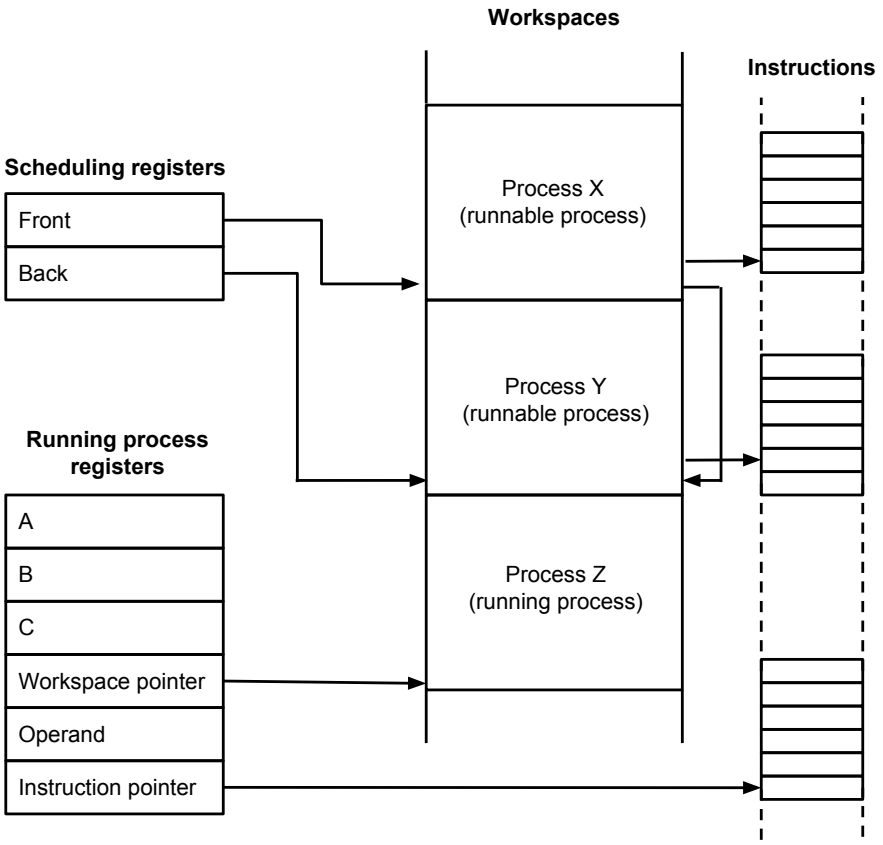

Figure 1. The Transputer's hardware implemented process scheduler [14].

A, B and C: these registers form the evaluation stack used to compute results for expressions.

Workspace pointer (Wptr): this holds the memory address of the top of the workspace stack.

Instruction pointer (Iptr): this stores the byte address of the next instruction.

Operand: the least significant 4 bits of the instruction are shifted into this register.

The Transputer can only execute a single process at a time, yet must keep track of all of them and schedule them on its CPU. Processes that are *runnable* (i.e. not waiting on channels and/or a timeout) – but not actually *running* – are queued in a linked list (the *run-queue*), with the processor maintaining pointers to the front and back nodes in registers. Each node in this list is the workspace of a runnable process (Figure 1). Processes that are not runnable, and not running, have pointers to their workspaces either in the channels on which they are *waiting* and/or the *timer-queue* (which is also routed through the waiting process workspaces). If an awaited event happens, the relevant process workspace node is placed at the back of the run-queue. When the currently executing process has to wait (for channels and/or timeout), its tiny run-context is saved in its workspace and the workspace pointer saved (in the channels and/or timeout-queue). The process at the head of the run-queue is then removed, its tiny run-context restored and execution resumed [14].

The fact that the Transputer automatically context-switches between its processes means that it implements a scheduler in hardware. In most architectures, these operations are entrusted to operating systems. However, the tight relationship between occam and the Transputer means that these operations are extremely efficient and, in many cases, are executed by a single assembly instruction.

1.3. Inter-process Communication

Two processes can communicate by using the occam channel primitives that are directly operated by Transputer instructions. The processes might reside within the same or different machines, yet the same instructions are used. In the former case, a channel is represented by a word in memory. When the first process becomes ready, it writes its workspace pointer (identity) in the channel and is descheduled. Then, when the second process is ready the message is copied by the processor to the specified location, the first process rescheduled and the channel returned to the empty state. On the other hand, if the processes reside in different Transputers both sender and receiver are descheduled while the transfer takes place and rescheduled when it concludes. In this case the communication is performed by autonomous controllers that operate the external links to other processors [15].

2. OpenTransputer System

The OpenTransputer comprises five major components as shown in Figure 2. The CPU is the main component that executes all core instructions as well as internal communication operations. It is also responsible for interfacing and managing all other components of the system and ensuring that all processes are scheduled correctly and fairly on its CPU. The input and output link controllers run concurrently and independently of the CPU. The links interact with the processor to enable communication operations between processes that reside in different CPUs. Moreover, the link controllers interface with network switches to pass messages between OpenTransputers.

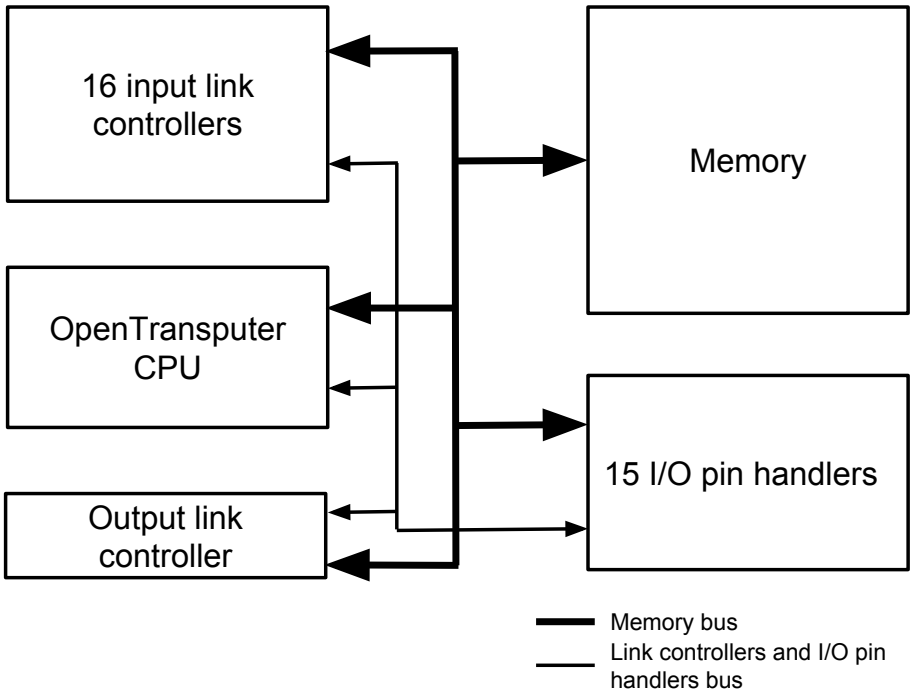

Figure 2. Major components of the OpenTransputer system.

There are 15 I/O pin handlers that interact with the processor in the same fashion as the link controllers. Nevertheless, instead of communicating with other processors, the handlers have a direct connection to the external I/O pins where hardware peripherals can be connected. The final major component of the OpenTransputer system is 10KB of on-chip Random Access Memory (RAM). The memory gives access to 32-bit words per operation,

and contrary to the original Inmos design, addresses start from 0 rather than from the most negative integer [15]. The CPU has a direct connection to the memory, while communication links and I/O pin handlers access it by means of a Direct Memory Access (DMA) controller. In the interest of prioritising communication operations over individual process execution, the DMA controller has a higher priority to access memory than the CPU. The remaining of this section describes the design and implementation of each major component of the OpenTransputer.

2.1. OpenTransputer CPU

The CPU comprises three major components as shown in Figure 3. The fetch unit is the simplest and is in charge of retrieving instructions from memory and feeding them to the control unit where they are associated with a microinstruction and executed. In contrast, the datapath is a collection of modules such as register files, Arithmetic Unit (AU), Logic Unit (LU) and memory addressing units all connected together that actually execute the instructions. Finally, the control unit is the authority in the processor and is the piece of hardware that generates the signals to enable or disable different paths within the machine, yet it does not directly interface with memory. We now describe each of these components in detail.

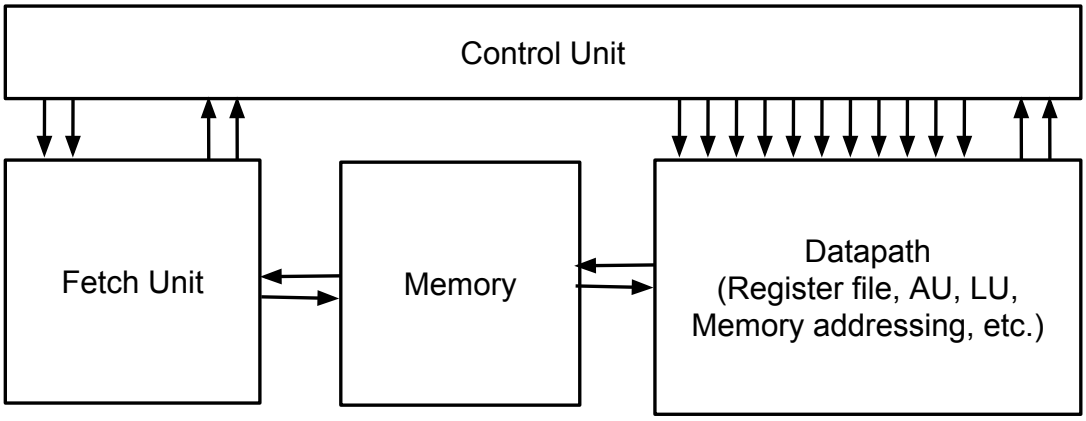

Figure 3. Integration of the three main components of the OpenTransputer CPU.

2.1.1. Control Unit

The original Transputer used a microcode system to generate the control signals needed to drive the datapath at each clock cycle. These signals were stored in Read-Only Memory (ROM) and loaded by a microcode engine when required. The alternative approach is to use *hardwired* logic that generates the same control signals by implementing combinatorial circuits with a few sequential elements. Considering that in the most complex Transputers there are over 600 microinstructions each consisting of more than 100 bits, hardwired logic would have significantly increased the area and complexity of the design.

An attractive feature of microcoded over hardwired systems is that they are much simpler to debug and change, even in later stages of the design process. This is because in these systems most of the machine's behaviour is described by microcode routines rather than by dedicated circuitry. Indeed, the development of these routines resemble programming assembly code rather than hardware design. The disadvantage is that microcoded processors that store their routines in ROM are potentially slower than their hardwired counterparts because there is a delay associated with reading the control signals from ROM.

For the OpenTransputer, we have decided to take a middle-ground between the microcoded and hardwired approaches by taking the best features of both. We maintain the flexibility and ease of use of the microcodes and the performance of hardwired logic. Hence, we devised a simple assembly-like language to describe the microinstructions, and a microassembler was developed in Python. The script parses the microinstructions in plain text and converts them into an intermediate representation that can be easily translated into any implementation such as ROM or hardwired logic. Therefore, when a bug in the design is found or a new feature is introduced, only the source microinstruction is modified and a new version of the control unit can be generated with a few keystrokes.

The OpenTransputer microassembly language was inspired by the original Transputer microinstructions. Each command in the microprogram can be thought of as a different state in a Finite State Machine (FSM) having an integer identifier (address) and an associated set of control signals [16] which are generated by the Python script. We describe microinstructions to the tool by using high-level human-readable text commands. Each microinstruction has three parts:

State name: this is an alphanumeric word that uniquely identifies the state.

Body: this describes the datapath enabled by the control unit in a single clock cycle. Common commands used in the body of a microinstruction select inputs from multiplexers, generate specific Arithmetic Unit (AU) and Logic Unit (LU) operation codes, enable registers for writing, etc.

Control flow: at each clock cycle, the CPU must be able to decide what happens next. Hence, all microinstructions include a bit field that encodes what action should be taken next. There are four possibilities:

1. Execute the next instruction from the instruction buffer (see Section 2.1.3).
2. Fetch the next instruction from memory. Only occurs when a microinstruction manipulates the instruction pointer, such as in jumps and context switches.
3. Unconditionally execute the next microinstruction whose address is embedded within this state.
4. Jump to one of two or one of four states according to a condition specified by this microinstruction.

An example microinstruction with state name `GT` is shown in Listing 1. The body compares the values of A and B registers using the AU; the commands `Auop0fromB` and `Auop1fromA` select the output of those registers as the input operands to the AU. The result of the computation is stored in A and the register stack is popped using the commands `AfromAubool` and `BfromC` respectively. Also, the operand register is cleared (set to 0) by the command `OfromClear`. Finally, the control flow command `Gotoplus1` increments the instruction pointer by one byte and tells the processor to execute the next instruction.

```
GT BfromC
   AU(gt)
   AfromAubool
   Auop0fromB
   Auop1fromA
   OfromClear
   Gotoplus1;
```

Listing 1. OpenTransputer microinstruction for state `GT` (greater than instruction).

The microinstruction `CCNT11` in Listing 2 performs an AU operation similar to `GT`, but uses the boolean result of this computation to decide whether control should be transferred

to either `CCNT13` ($B < A$) or `CCNT14` ($B \geq A$). Clearly, these human-readable instructions can be easily translated into bit patterns by a microassembler.

```
CCNT11 Auop0fromB
       Auop1fromA
       AU(ulteq)
       Condaubool(CCNT13,CCNT14);
```

Listing 2. OpenTransputer microinstruction for state `CCNT11`.

2.1.2. Datapath

Due to constraints in the manufacturing process during the 1980s, its was difficult to fabricate chips with more than two layers of metal. Therefore, the number of connections between the different components of the processor had to be minimised to avoid issues such as wires crossing over each other's paths. This is the main reason why the original Transputer was implemented using three buses. Two of these buses contained the input operands for components such as the ALU. The third gathered the result of a computation and was also used to write the value back to its destination. Despite meeting manufacturing requirements, this design approach has the effect of reducing the number of simultaneous computations that can be performed at any clock cycle. This is due to the fact that it is not possible to transport the operands to the relevant components in parallel.

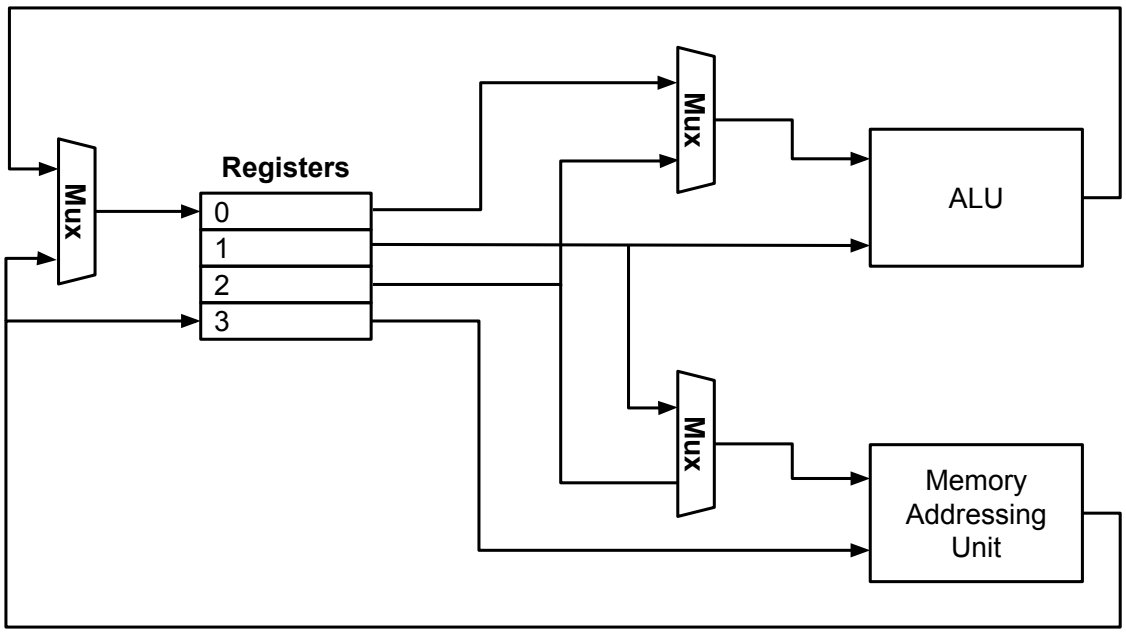

Figure 4. Example implementation using a *wide* datapath approach.

Modern manufacturing technique can easily cope with multiple layers of interconnect, giving us flexibility to include significantly more wiring. Hence, we have decided to replace the buses by large collections of wires that transport the output of a component to every part of the datapath that will eventually need that signal as an input as illustrated in Figure 4. The result is a *wider* datapath where it is possible to feed different input values to more modules of the datapath within the same clock cycle. This allows us to reduce the overall cycle count of many instructions since we are now able to complete more operations per cycle. For instance, we are now able to compute an addition using the ALU and a memory address

using the addressing unit simultaneously. To fully take advantage of our approach, we have also decided to replicate some of the functionality within the datapath such as the addressing unit. This allows us to compute the same operation on different operands simultaneously and increase the overall performance of the processor.

A final optimisation introduced in the OpenTransputer greatly reduces the time taken to perform a context switch between low and high priority processes. When a low priority process wishes to start a new high priority one, the original Transputer blocks the parent process by storing all its context in memory and then executes the new one. This task is particularly slow because the processor must write data to memory at least six times to store the A, B, C, Iptr, Wptr and status registers.

We have chosen to optimise context switches from low to high priority in the OpenTransputer by introducing *shadow registers* [17] within the register file. Shadow registers are additional registers whose only purpose is to temporarily save the value of another register that will be overwritten and would otherwise be lost. Thus, when a low priority process is blocked by a high priority one, the CPU copies all the context into the shadow registers in a single clock cycle. When the low priority process needs to be restored, the saved values are recovered from the shadow registers. This mechanism reduces the average context switch time from 12 to 4 clock cycles.

2.1.3. Fetch Unit

The fetch unit is very simple since the OpenTransputer does not have a conventional Fetch-Decode-Execute pipeline, a mechanism that is better suited to register machines rather than to stack-based architectures. The fetch unit consists of a 32-bit instruction buffer (that holds four 8-bit instructions) and some logic to decide when the next fetch should be issued. Its operation is also very simple and is tightly related to the control unit. Firstly, when the machine is switched on a fetch is issued before any instruction is executed. In normal operation the fetch unit uses the two least significant bits of the instruction pointer (the byte index) as an index into the instruction buffer to decide which instruction is executed next. When the byte index reaches 3 a new fetch is issued and execution proceeds. This mechanism does not take into account that instructions must be fetched when the instruction pointer is manipulated either by a jump or any other operation such as context switches.

2.2. OpenTransputer External Communication

2.2.1. Beneš Networks

The Inmos Transputer has four external bidirectional communication links that can be used to connect up to four devices. There is no limit in the number of Transputers that can be connected in this fashion [18]. However, as the network grows communication operations may be slower since messages need to be relayed by intermediate nodes before the destination is reached. This clearly reduces the usability of the Transputer as a building block for assembling large parallel networks. Inmos realised this problem and developed a 32-way crossbar switch known as C004 [19] that is compatible with the serial protocol of the Transputer links. The idea is that the switch routes messages between the Transputers rather than joining the processors using direct point-to-point connections.

Taking this into account, we implemented the OpenTransputer with a single (parallel) bidirectional link rather than four (serial). This link is meant to be connected to small switches arranged in a Beneš network. Beneš networks [20] are a special form of Clos networks [21] composed of simple 2×2 switches and three or more stages. They were formalised by Václav E. Beneš while working at Bell Labs researching different topologies for telephone switching networks.

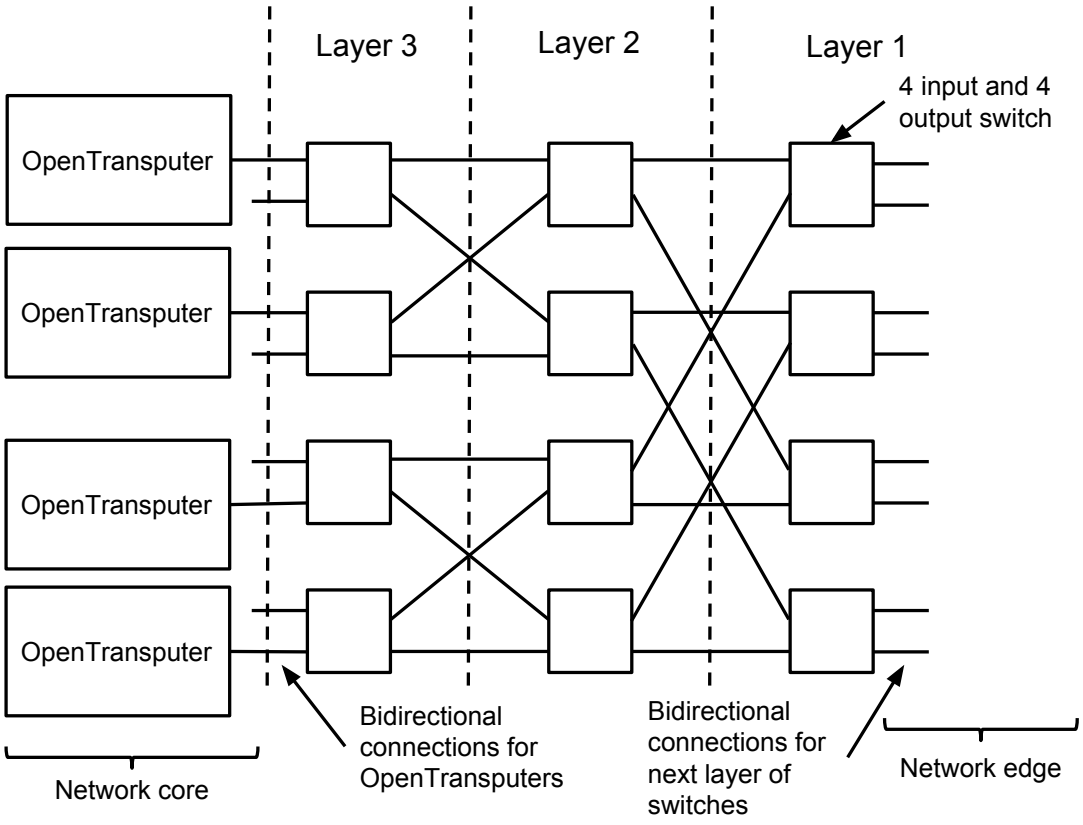

Figure 5. Folded over Beneš network with capacity for 8 OpenTransputers.

In a Beneš network, if r is the number of switches in every stage, the network supports $N = 2 \cdot r$ different inputs and the same number of outputs. Moreover, the network contains $2 \log_2 N - 1$ stages and a total of $N \log_2 N - N/2$ switches. In the case of the OpenTransputer, each processor needs both an input and an output link to communicate. Therefore, the Beneš network can be viewed as folded on itself as shown in Figure 5. The resulting Beneš network is composed of crossover switches with four inputs and four outputs (4×4) rather than the simpler 2×2 crossbar switch.

Beneš networks are rearrangeably non-blocking [20], meaning that given any input a path can always be found to an output without colliding with any other path from other connections in the network. However, in order for this to happen some of the previous connections might have to be rearranged to use different intermediate switches. For the OpenTransputer, the non-blocking property of Beneš networks means that given the communication pattern of a program, it is always possible to find an arrangement of the channels in the network such that no two paths cross. In other words, we can ensure that no two packets will try to use the same link when the program is being executed. Clearly, this is an attractive property because in concurrent systems the performance of a program is often limited by the network traffic. For instance, in a more traditional matrix arrangement, the shortest paths between two nodes inevitably use the switches in the main diagonal. As a result, the traffic through these nodes of the network is significantly higher causing a large number of collisions and slowing down the whole system.

The non-blocking property is the main reason for our choice of network configuration. The occam programming model facilitates analysing programs statically in order to extract the communication pattern and efficiently map channels to non-blocking paths in the network. It is important to highlight that this requires the development of new software tools that generate such mappings depending on predefined network topologies.

2.2.2. Channel Addressing

In the OpenTransputer, channels may be used for internal and external communication and I/O pin handling. Since the same input and output instructions are used for all these operations, the processor uses channel addresses to differentiate what actions should be performed. As illustrated in Figure 6, the least significant two bits of the address are used to identify the type of channel. Hence, if bit 0 is set, the processor knows that this is an external channel address and the I/O instructions should be executed accordingly. On the contrary, if the least significant bit of the address is not set, the decision between internal communication and I/O pin handling relies on the value of the bit with index 1.

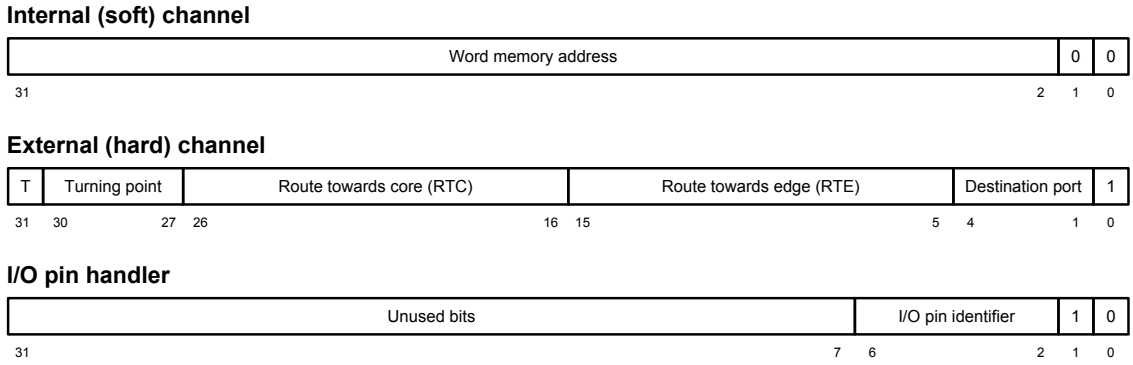

Figure 6. Bit fields of the channel addresses for internal and external communication and I/O pins.

For internal communication, any word in memory can be used as a channel: the channel address is a memory address with the least significant two bits set to 00. For external communication, the channel address is odd and encodes the route followed by a packet through the Beneš network to reach the destination OpenTransputer. For an I/O pin, the channel address ends with 10 and encodes the identifier of the pin to be used.

2.2.3. Message Routing

The Beneš network is implemented as a collection of smaller four input and four output switches. As shown in Figure 5, the OpenTransputers are connected at one end of the network that we call the *core*. The other end of the network of switches always ends in unconnected ports and is called the *edge*. The available connections in the edge can be used to attach more switches and increase the capacity of the network. Every new layer of switches linked into the edge doubles the capacity. The routing operation within the network of switches consists of two stages. Firstly, the message is forwarded by the individual switches towards the edge of the network. Then when the message reaches a pre-determined layer (*turning point*), it is turned back towards the core and its destination. Routing through the layers is steered by an 11-bit field in the channel address – one for each direction (Figure 6). Thus, a maximum of 11 layers may be built, allowing up to 2048 OpenTransputers to be connected.

Messages are either *application data* or *acknowlegements* (for end-to-end flow control). A message is sent through the network as a pipeline of *packets*, where a packet contains just a single byte of the message (plus tag bit indicating whether application data or acknowlegement) together with its target channel address (that contains the route). Packets are moved through, and between (if on the same piece of silicon), switches in *parallel* (over 41-bit wide paths: 32-bit address, 8-bits of data and 1-bit for control flow). Details may be found in the dissertation available from [8].

An example of routing is shown in Figure 7. The network consists of 12 switches and two OpenTransputers connected on opposite ends of the network. The sending OpenTransputer

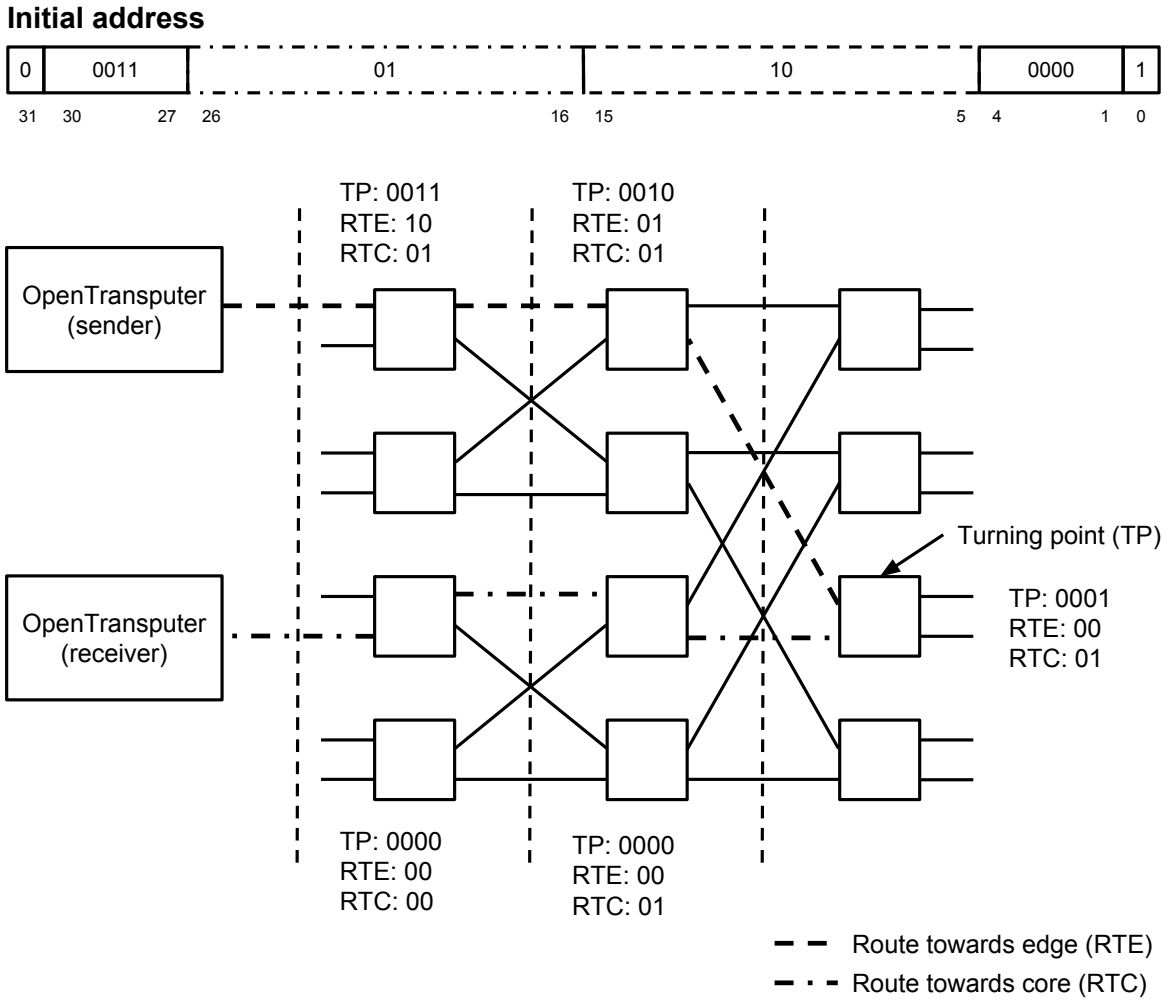

Figure 7. Example route of a packet through a Beneš network of OpenTransputers. The information shown next to each switch is the state of the packet address at that point of the network.

sends a packet to its connected switch. In this example, since the *Turning Point* (TP) field of the channel address initially in the packet is greater than 0001, the packet has not reached its turning point and the switch inspects the least significant bit in the *Route Towards Edge* (RTE) field. Because this bit is not set, the switch forwards the packet to the next switch in the path using its *left* output port. Before forwarding this packet, the switch modifies the channel address by decrementing the TP field and shifting out the already-used bit from the RTE.

The second switch inspects the address and modifies it in the same fashion; this time it forwards the packet using its *right* output port because the least significant bit in the incoming RTE was 1. The third switch in the path finds the TP field to be 0001, indicating that it is the turning point. This switch decrements the TP field to zero and reflects the packet back, using the port on which it did not arrive.

The next switch in the network sees a zero TP field, so checks the *Route Towards Core* (RTC) field and, since the least significant bit is 1, forwards the packet using its *right* output port (having first shifted out the used RTC bit, with the TP left at zero). This process continues until the packet reaches the destination OpenTransputer.

Each OpenTransputer is connected to the network by a single bidirectional link. A processor receiving a packet carrying application data is responsible for delivering it to the appropriate input link controller (specified by the destination port in the channel address). A processor receiving an acknowledgement packet delivers it to its output link controller.

2.2.4. Autonomous Link Controllers

The OpenTransputer has a single bidirectional link that is connected to a network switch. The link is shared between the output and input link controllers that were implemented using exactly the same microcoded approach explained in Section 2.1.1. In this section we describe the operation of the controllers and their interaction with the CPU.

Output controller. A disadvantage of the original Transputer design is that only one external output and one external input channel can be bound to a hardware link (and both must start and end with the same pair of Transputers). Therefore, there can be only 4 external output channels and 4 external input channels connected at any time, though all can operate concurrently. If an application needs more external connections than this, the necessary multiplexing and routing software must be installed, the application must be coded to use it and the Transputer CPUs must execute it (taking time away from application compute code). To mitigate this problem, the OpenTransputer provides hardware support for *virtual channels*. These are simpler, though not so powerful, as those introduced in the T9000 Transputer [22] and, currently, only support external output channels.

Virtual channels are an abstraction that gives the programmer the impression that there is an infinitely large number of output links. In reality, there is a single link that is shared between all communicating processes. For this purpose, we developed an output link controller that implements a scheduler in hardware similar to that used by the CPU to allocate processing resources. The output link controller has a pair of registers that store pointers to the front and back processes of a linked list. This structure acts as a queue where new processes are appended to the back and dequeued from the front to be given a share of communication time. It is important to highlight that currently both high and low priority processes are queued in the same linked list (i.e. priority is ignored in the ordering of messages ready to go – first ready, first sent). Furthermore, an on-going communication cannot be interrupted: if a low priority process is using the link and a high priority process wishes to communicate, the latter will be appended to the back of the queue and dealt with only when communication operations from processes previously appended completes. These are clear limitations of the current implementation, and will be overcome in future releases of the OpenTransputer design (e.g. by maintaining separate queues for high and low priority processes waiting to send).

The operation of the output controller is conceptually simple. When the OpenTransputer is reset, the queue is emptied and the controller set to its ready state. When a process wishes to output a message, the CPU sends the size and pointer to the message and the workspace pointer to the link controller. Since the link is not busy, the output operation is run by the controller. If another process wishes to use the link while it is busy, the controller uses the DMA mechanism to store the length and pointer to the message in the process' workspace and enqueues the process. When a communication completes, the output link controller sends a signal to the CPU to request that the responsible process be scheduled for execution. Then, the controller removes the next waiting process at the front of the queue (if any) and runs the output operation – the length and pointer to the message it has to send are available in the process' workspace. If the queue is empty, the controller waits. It cannot be stressed enough that the output link controller runs concurrently and independently of the CPU: while the controller is performing an operation, the CPU will be executing another process.

Input controllers. It is difficult to implement the idea of virtual channels for input operations. This is because if we modify the input link controller to implement a queue

of processes, then we would need a buffer of potentially infinite size to store the partially received messages from other OpenTransputers. Furthermore, messages from any waiting process can arrive at any point of time. Thus, maintaining a simple linked list of all processes would be very inefficient, since in the worst case the controller would have to iterate through every item in the list to match an incoming message with its intended recipient. In this sense, a more sophisticated data structure should be used to implement the scheduler of the input link controller. However, complex schemes are difficult to implement in hardware and have other undesirable side effects such as increased power consumption.

Therefore, we consider that implementing an input controller similar to the output one has more disadvantages than benefits – but we also consider that four links, as in the original Transputer, are too limiting. For this reason, we decided to include 16 input link controllers per OpenTransputer, all sharing the single bidirectional link with the output controller.

Each of the 16 input link controllers corresponds to a different port number as specified in the channel address for external communication (Figure 6). Similar to the output controller, the input controllers use the DMA mechanism to access memory and operate concurrently and independently from each other and the CPU.

The interaction between CPU and input link controllers is slightly more complex than that for the output link controller. This is because in occam input operations can be used within *alternative* constructs. The controller can be in any of four states:

Ready. Data (first byte) has been received from a remote process, but there is currently no receiver.

Requested. The controller is communicating on behalf of a process.

Enabled. In an occam alternative construct, processes are bound to guards (with optional pre-conditions). Only a process whose guard is *ready* (and pre-condition, if present, is *true*) can execute and only one such will be selected to execute. Channel inputs are the most common form of guard and they become ready when a message is pending. To support alternative constructs including *external* channel input guards, the relevant input link controllers must be set to *enabled*, meaning that they must notify the CPU they are ready if and when the first byte of a message arrives. If such an input is selected by the CPU, the relevant controller will be told to accept the rest of the message. Otherwise, it will be *disabled* and will refuse reception of the rest of the message (by not acknowledging that first byte).

Waiting. The controller is not communicating on behalf of a process and no data has arrived in the link.

When the controller is reset, its state is set to waiting. If a process wishes to execute an input operation, the CPU interacts with one of the input controllers by sending it the message length and destination pointer as well as the process' workspace pointer. The input controller will become *requested* and run this operation to completion. When the message has been fully received, the CPU will be flagged to reschedule the process. On the other hand, if data is received before the controller is *requested*, it will enter a *ready* state and wait until a process performs an input operation on that controller. Afterwards, the communication continues as normal. The final case is when the CPU executes an alternative construct. In this case, all involved input controllers are *enabled*: one of them notifying the CPU that data has been received *may* be allowed to communicate, depending on the guard selected by the CPU. The remaining enabled controllers are set to *ready* or *waiting* state (disabled) depending on whether they have received data from the link.

2.2.5. Flow Control

The network must provide end-to-end synchronisation for all messages between sending and receiving processes. Also, since the network provides minimal buffering, it is essential that no packet is launched without knowing that the input controller on the destination OpenTransputer is ready and committed to take it – otherwise, the network will quickly become jammed.

To send a message, therefore, the output controller sends the first packet only and waits for an acknowledgement packet. The destination input controller (which will be waiting) has room to store the first byte of an incoming message – so that packet will be accepted. If it has been told by the CPU that an application process is committed to receive, an acknowledgement packet is returned after the received data is stored in memory. If not, it waits until it is so told before storing the data and sending the acknowledgement.

As part of its instruction from the CPU, the input controller now knows the message length (in bytes) and where (in the receiving process' workspace) to store that first byte received – and all remaining bytes. When the output controller receives the acknowledgement (of its first packet), it sends the second data packet and waits for an acknowledgement. This process continues until all remaining data is transmitted. No special packet is sent by the output controller to signal the end of communication, since the input controller is notified of the message length in advance by its own CPU.

However, we understand that in some applications performance is key and the message length is relatively large. For this reason, in future iterations of the design, it is desirable to introduce special *streaming* channels. These shall require a single acknowledgement from the input controller for the first packet, yet the remaining packets are *streamed* – sent by the output controller without receiving acknowledgements from the remote peer. Note that this preserves the existing property (for *unstreamed* channels) that there is no limit in the message length that can be sent. The first packet acknowledgement only happens when the receiving OpenTransputer has been told (by its CPU) the size of the incoming message and where to store it – just the same as for an unstreamed channel. So, no additional buffering for the message is needed by the receiving processor and any message length can be accommodated. We have designed the OpenTransputer to facilitate the development of this feature in the future; for instance, the reserved bit of the external channel address (bit 31 in Figure 6) is intended to signal a streaming channel.

2.3. OpenTransputer I/O Pins

In many modern processors, input/output (I/O) pins are accessed through a mechanism known as Memory-Mapped I/O. Certain locations in memory are reserved and correspond to I/O pins. Communication with I/O devices is handled through these reserved memory locations. When an event occurs in any of the I/O pins, an interrupt diverts the processor's attention to the event, so that it can be handled without delay.

It could be argued that this approach is difficult to grasp for the inexperienced (and also experienced) user. Such an approach is conceptually different from the channel mechanism found in the Transputer and OpenTransputer.

The OpenTransputer treats I/O pins in the same way external communication links: each pin handler is a special, but very small, link controller. Software access to the pins has the same syntax and semantics as communication over any kind of of channel: a process listening or writing to a pin waits for the pin handler to complete the operation, thus synchronising with whatever external hardware is driving the pins. So, all we need is standard occam code (*channel I/O*) to synchronise and communicate with peripheral hardware such as sensors or displays.

The I/O pin handlers in the OpenTransputer expose the same interface as the communication links. In other words, the processor interacts in exactly the same way with both the pin handlers and the autonomous link controllers. When the processor wants to read or write a pin, it sends the handler the number of bytes to be read or written and the memory address to where the data should be stored in or read from. The process that is waiting for I/O is descheduled and enters a waiting state. When the pin handler finishes the requested operation, it signals the processor, which then reschedules the process and execution proceeds as normal.

The I/O pin handlers are microcoded components in exactly the same way as the communication link controllers and the CPU, yet much smaller. We have chosen a microcoded approach to guarantee that the I/O pin handlers can be easily modified in the future to add new functionality. For additional flexibility, a new instruction, `confio`, has been introduced to configure/re-configure the I/O pin handlers at runtime. When executed, the B register holds the address of the target I/O pin handler with the required configuration in the A register.

Channels are used for three kinds of synchronised communication: between processes on the same OpenTransputer (*soft*), between processes on different OpenTransputers (*hard*) and between a process and the I/O pin handlers of an OpenTransputer. The format of channel addresses (Section 2.2.2) was designed to distinguish between these three kinds of use. For I/O pin handlers, the third form in Figure 6 is used. The I/O pin handler identifier is stored in five bits (2..6): the input link identifiers are 0 through 15 and the output link is 16, leaving 17 through 31 for the fifteen I/O pin handlers.[1]

We cannot stress enough that the current implementation of the I/O pin handlers is extremely limited and is intended for demonstration purposes only. For instance, it is only possible to set a pin in input or output mode. In the former case, when the processor makes a request, the pin handler simply reads the input pins and stores each byte in memory using the DMA controller. If the pin handler is configured as output, when a request is made to the handler, it reads the memory location where the data to output is stored and places it in a register within the pin handler. Another important limitation is that the I/O pin handlers will only output or input the least significant bit of the first byte of data, rather than the total number of bytes requested by the processor as described above. This is mainly because there is no mechanism to set the I/O pin handler's input or output rate, which could be achieved by synchronising the handler with a specific clock source. Furthermore, the current implementation does not allow an I/O pin handler to drive more than a single physical pin (so that, for example, a byte could be sent received across 8 pins in parallel).

3. Critical Evaluation

Since the OpenTransputer is still in its early stages of development, it is difficult to make an objective comparative study with other architectures. In contrast, we compare the performance of the OpenTransputer with the Transputer in terms of cycle counts. We also discuss the result of the synthesis process and mention prominent findings.

3.1. Comparing OpenTransputer and Transputer

We will compare the performance of our implementation with that of the Transputer. Our metric is the number of cycles taken to execute the core instructions in both the OpenTransputer and the Transputer. Table 1 shows the cycle count for the *primary* ([13]) instructions.

The table shows that, for 6 of the 15 instructions, the number of cycles improved by at least a factor of 2. We consider that this is the effect of the significant changes introduced

[1]To simplify the silicon design, the CPU communicates with the I/O pin handlers via the same 5-bit wide bus used for communicating with the link controllers (shown in Figure 2). This is why the identifiers for the link controllers and I/O pin handlers are fitted into the same 5-bit address space.

Table 1. Execution time (in clock cycles) of primary instructions: Inmos Transputer and OpenTransputer

Instruction	Inmos Transputer	OpenTransputer
ldl	2	1
stl	1	1
ldlp	1	1
ldnl	2	1
stnl	2	1
ldnlp	1	1
eqc	2	1
ldc	1	1
adc	1	1
j	3	1
cj	2-4	2
call	7	4
ajw	1	1
nfix	1	1
pfix	1	1

in the datapath of the processor with regards to module replication and additional wiring between the components.

Notice that the scheduling instructions (i.e. `runp` and `startp`) have improved by a factor of 2 in the worst case. This is due to the fact that we have implemented shadow registers to improve the performance of context switches between high and low priority process as discussed in Section 2.1.2. The use of shadow registers eliminates the need to perform 6 memory accesses to store the context of the process being blocked. Therefore, the worst case scenario for `startp` and `runp` is reduced to only 5 clock cycles, while the Inmos Transputer takes approximately 12.

A final observation is that the performance of memory block transfer operations has decreased by approximately a factor of 2. Recall that the OpenTransputer contains dedicated combinatorial logic blocks that implement most of the functionality pertaining these operations. However, it can be inferred from the table that the equivalent logic in the Inmos Transputer is significantly faster and is only bound by memory since there are two accesses per word transferred. This is due to the lack of optimisations in the current OpenTransputer design and would be part of future development on the platform.

3.2. Synthesis Results

We developed a prototype implementation of the OpenTransputer in Verilog HDL. The design consists of a network of two processors connected by a switch. We synthesised the design for two different targets, a ZedBoard XC7Z020-CLG484 Field-Programmable Gate Array (FPGA) and a silicon target for a manufacturing process.

3.2.1. FPGA Synthesis

An FPGA uses Look-Up Tables (LUT) to implement logic. A LUT can be described as having an arbitrary number of inputs and one output. It can then be programmed to assume a certain output value depending on the inputs. The synthesised OpenTransputer prototype runs at 41 MHz and utilises 35% of the FPGA's LUTs. Most LUTs are used as logic (86%) and only 14% are used as memory. Furthermore, in comparison to the processors, the resources consumed by the communication switch are negligible.

Examining the utilisation of LUTs for a single OpenTransputer, we observed that the datapath consumes most of the resources. It uses 79% of all the LUTs and 57% of all the registers inside the processor.

As the datapath is such a major component of the processor, Table 2 shows which of its parts use the largest number of resources in the FPGA. Surprisingly, the autonomous controllers use over half of all the LUTs and almost 90% of registers in the datapath. The RAM on the other hand contains all the LUTs used as memory within the processor, yet very little logic. The register file, which contains the registers accessible to the programmer uses 19% of all the LUTs and 9% of all the registers in the datapath.

Table 2. FPGA resources used by the major components within the datapath

Component	LUTs	Registers	LUTs as logic	LUTs as memory
Datapath	7,372	5,334	6,060	1,312
Autonomous links	4,061	4,599	4,061	0
RAM	1,484	0	172	1,312
Register file	1,408	503	1,408	0
Other	419	232	419	0

The reason why the autonomous controllers use so many resources can be explained by their sheer numbers: there are one output and 16 input controllers per processor (without counting I/O handlers), more than twice as many than in the Inmos Transputer. Each controller encompasses the microcode sequencing logic similar to that of the processor's control unit and three different interfaces: DMA controller, the physical link and the CPU. It is important to highlight that these components can be greatly optimised in future versions of the OpenTransputer.

3.2.2. Manufacturing Process Synthesis

In some respects, the OpenTransputer is more complex than the Transputer since it uses a *wide* datapath with multiple instances of the same logic modules. On top of this, the OpenTransputer is still in its early stages of development and is not fully optimised, but due to technological advancements in the last two decades we expect some sort of relation.

Table 3. Comparison between OpenTransputer and the original Transputer after synthesis

	OpenTransputer	Transputer
Chip Area	3.69 mm^2	64 mm^2
Manufacturing process	180 nm	1000 nm

Moore's law refers to the observation that in the history of modern computing the number of transistors in an integrated circuit doubles every two years [1]. Keeping this mind we can make some interesting observations about the synthesis results listed in Table 3. We see that the area of the OpenTransputer is 3.69 mm^2 while the Transputer in 1985 had an area of approximately 64 mm^2; in other words, there is a decrease in area by a factor of 17.3. Since the OpenTransputer is more complex than the original Transputer, i.e. is made up of more hardware components and by extension transistors, an explanation for the area reduction is that the individual components have shrunk in size. As we see in the second row of Table 4.6, this has indeed been the case, the OpenTransputer is targeted at 180 nm technology, while the Transputer has been targeted at 1000 nm. This implies a reduction in the size of transistors by a factor of 5.6.

If the OpenTransputer and Transputer were completely identical in terms of transistor count, then, according to Moore's law, the decrease in area by a factor of 17.3 would only

be due to a reduction of the target technology by the square root of this factor, i.e. $\sqrt{17.3}$ or roughly 4.2. This implies that (to some extent) the area reduction of the OpenTransputer with regards to the Inmos Transputer follows Moore's law. The difference between the technology reduction factor (4.2) and the total area factor (5.6) can be attributed to the lack of optimisations and the more complex microarchitecture of the OpenTransputer.

4. Future Work

The primary focus of our efforts so far has been the development of the three basic components of the OpenTransputer: CPU, external communication infrastructure and I/O interface. However, we envision the OpenTransputer as an environment of software tools and hardware components to enable other developers to integrate the processor within their products. In this sense, the scope of future work is potentially unbounded, yet we propose a number of directions to explore:

- A software development environment (compiler, loader, debugger, etc) that facilitates the use of the OpenTransputer is paramount if the device is intended to be used for commercial purposes. Currently, our design does not have any practical means of loading and debugging programs on the actual hardware.
- Clearly, there are a number of components (co-processors) missing such as floating-point unit and memory interfaces. To facilitate the process of integrating these components, a well defined interface must be implemented that enables the flow of information between the CPU and the co-processors.
- The OpenTransputer can be easily used as a building block to assemble multicore systems, yet a serial communication protocol must be implemented in the switches for off-chip connections for the device to be practical.
- The link controllers allow multiple OpenTransputers to communicate over the Beneš network. Nevertheless, as mentioned in Section 2.2.5, there is room for improvement by changing the flow control mechanism to support streaming channels. Furthermore, the output link controllers must be modified to support interleaved outputting of messages. Currently, the link controller waits for a message to be fully transmitted before the next message is output. Clearly, this is a problematic if the receiving process is unable to accept the message since the output link controller could potentially wait forever.
- As described in Section 2.3 we have developed an early version of the I/O pin handlers that does not offer the necessary fuctionality for complex applications. For instance, it is not possible for a single I/O pin handler to drive more than a single pin physical simultaneously. However, we have designed the OpenTransputer in a modular fashion and introduced a new configuration instruction to facilitate future enhancements.

5. Conclusion

We have developed a new implementation of the Transputer architecture that we call OpenTransputer. We have designed a radically different microarchitecture that takes advantages of state-of-the-art manufacturing techniques and current developments in the field of computer architecture. For instance, the OpenTransputer CPU is implemented by using hard-coded logic rather than the microcoded ROM used in the Transputer. Furthermore, we replaced the four serial communication links by a single bidirectional parallel connection to a network of switches. These networks are arranged in a Beneš fashion providing rearrangeably non-blocking communication between all OpenTransputer nodes.

We have also developed a basic I/O interface built upon the channel communication functionality. This means that even the most simple occam program that simply outputs an integer to a channel can drive hardware peripherals.

With the rising popularity of new ideas such as IoT and the current state of the technology landscape, we consider that there is an opportunity for an open-source processor such as the OpenTransputer to be used in many applications.

Acknowledgements

First and foremost, we would like to express our gratitude to our supervisor, Prof. David May, for his valuable guidance and support throughout the project. Without his patient explanations about the Inmos Transputer and his advice our work would not be completed. Furthermore, we would like to thank Roger Shepherd for his early input and advice on how to develop the OpenTransputer. Thanks are also owed to Dr. Simon Hollis for his advice on digital design using the Vivado Design Suite and Fred Barnes for his input on the compiler and configuration system of the Transputer.

We would also like to thanks Richard Grafton and the University of Bristol Computer Science Department who supplied us with the necessary resources used throughout the project.

References

[1] Gordon E Moore et al. Cramming More Components onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[2] Mark Homewood, David May, David Shepherd, and Roger Shepherd. The IMS T800 Transputer. *Micro, IEEE*, 7(5):10–26, 1987.

[3] Inmos Limited. *Transputer Reference Manual*. Prentice Hall International (UK) Ltd., 1988. ISBN: 0-13-929001-X.

[4] B Tatry and M-A Claire. Myriade Microsatellites: a New Way for Agencies and Industry to Various Missions. In *Small Satellites, Systems and Services*, volume 571, page 4, 2004.

[5] HR Arabnia. The Transputer Family of Products and their Applications in Building a High Performance Computer. *Belzer, J., Holzman., AG, Kent, A.(eds.) Encyclopedia of Computer Science and Technology*, 39:283, 1998.

[6] Anthony JG Hey. *Supercomputing with Transputers—Past, Present and Future*, volume 18. ACM, 1990.

[7] HETE-2 Spacecraft [Online], Available: `http://space.mit.edu/HETE/spacecraft.html` [07 May 2015].

[8] OpenTransputer Home Page [Online], Available: `http://www.opentransputer.org/` [03 December 2015].

[9] David May and Roger Shepherd. occam and the Transputer. In *Proc. of the IFIP WG 10.3 Workshop on Concurrent Languages in Distributed Systems: Hardware Supported Implementation*, pages 19–33, New York, NY, USA, 1985. Elsevier North-Holland, Inc. `http://dl.acm.org/citation.cfm?id=2957.2959` .

[10] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[11] JR Newport. An Introduction to occam and the Development of Parallel Software. *Software Engineering Journal*, 1(4):165–169, 1986.

[12] Andrea Clematis and Ornella Tavani. An Analysis of Message Passing Systems for Distributed Memory Computers. In *Parallel and Distributed Processing, 1993. Proceedings. Euromicro Workshop on*, pages 299–306. IEEE, 1993.

[13] Inmos Limited. *Transputer Instruction Set: A Compiler Writer's Guide*. Prentice Hall, July 1988.

[14] David May. The Transputer Implementation of occam. Institute for New Generation Computer Technology, Presented at the International Conference on Fifth Generation Computer Systems, Tokyo, Japan, November 1984.

[15] Roger Shepherd. *Transputer System Description*. Inmos Limited, September 1988.

[16] M Tanaka, N Fukuchi, Y Ooki, and C Fukunga. Design of a Transputer Core and its Implementation in an FPGA. In *Proceedings of Communication Process Architectures 2004*, pages 361–372, Amsterdam, The Netherlands, 2004. IOS Press. ISBN: 1 58603 458 8. `http://www.wotug.org/paperdb/send_file.php?num=125` .

[17] Jagan Jayaraj, Pravin Lawrence Rajendran, and Thiruvel Thirumoolam. Shadow Register File Architecture: a Mechanism to Reduce Context Switch Latency. *College of Engineering Guindy, Anna University, Chennai, India*, 2002.

[18] Inmos Limited. *Transputer Databook*. Inmos Limited, 1989.

[19] SGS-Thomson Microelectronincs. *IMS C004 Programmable Link Switch*, September 1995.

[20] Václad E Beneš. Optimal Rearrangeable Multistage Connecting Networks. *Bell System Technical Journal*, 43(4):1641–1656, 1964.

[21] Charles Clos. A Study of Non-Blocking Switching Networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[22] M.D. May, P.W. Thompson, and P.H. Welch, editors. *Networks, Routers and Transputers: Function, Performance and Applications*. IOS Press, Amsterdam, The Netherlands, 1993. ISBN: 90-5199-129-0. `http://www.transputer.net/ibooks/90-5199-129-0/` .