Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

R16: a New Transputer Design for FPGAs

John JAKSON
Marlboro MA, USA
johnjakson@usa.com, transputer2@yahoo.com

Abstract. This paper describes the ongoing development of a new FPGA hosted Transputer using a Load Store RISC style Multi Threaded Architecture (MTA). The memory system throughput is emphasized as much as the processor throughput and uses the recently developed Micron 32MByte RLDRAM which can start fully random memory cycles every 3.3ns with 20ns latency when driven by an FPGA controller. The R16 shares an object oriented Memory Manager Unit (MMU) amongst multiple low cost Processor Elements (PEs) until the MMU throughput limit is reached. The PE has been placed and routed at over 300MHz in a Xilinx Virtex-II Pro device and uses around 500 FPGA basic cells and 1 Block RAM. The 15 stage pipeline uses 2 clocks per instruction to greatly simplify the hardware design which allows for twice the clock frequency of other FPGA processors. There are instruction and cycle accurate simulators as well as a C compiler in development. The compiler can now emit optimized small functions needed for further hardware development although compiling itself requires much work. Some occam and Verilog language components will be added to the C base to allow a mixed occam and event driven processing model. Eventually it is planned to allow occam or Verilog source to run as software code or be placed as synthesized co processor hardware attached to the MMU.

Keywords. Transputer, FPGA, Multi Threaded Architecture, occam, RLDRAM

Introduction

The initial development of this new Transputer project started in 2001 and was inspired by post-Transputer papers and articles by R. Ivimey-Cook [1], P. Walker [2], R. Meeks [3] and J. Gray [4] on what could follow the Transputer and whether it could be resurrected in an FPGA. The conclusion by J. Gray was that it was a poor likelihood; he also suggested the 4-way threaded design as a good candidate for implementation in FPGA. In 2004 M. Tanaka [5] described an FPGA Transputer with about 25 MHz of performance, limited by the long control paths in the original design. By contrast DSPs in FPGA can clock at 150 MHz to 300 MHz and are usually multi-threaded by design. Around 2003, Micron [6] announced the new RLDRAM in production, the first interesting DRAM in 20 years. It was clear that if a processor could be built like a DSP, it might just run as fast as one in FPGA.

It seems the Transputer was largely replaced by the direct application of FPGAs, DSPs and by more recent chips such as the ARM and MIPS families. Many of the original Transputer module vendors became FPGA, DSP or networking hardware vendors. The author concludes that the Transputer 8-bit opcode stack design was reasonable when CPUs ran close to the memory cycle time but became far less attractive when many instructions could be executed in each memory cycle with large amounts of logic available. The memory mapped register set or workspace is still an excellent idea but the implementation prior to the T9000 paid a heavy price for each memory register access. The real failure was not having process independence. Inmos should have gone *fabless* when that trend became clear, and politics probably interfered too. Note that the author was an engineer at Inmos during 1979-1984.

In this paper, Section 1 sets the scene on memory computing *versus* processor computing. Section 2 gives our recipe for building a new Transputer, with an overview of the current status on its realization in Section 3. Section 4 outlines the instruction set architecture and Section 5 gives early details of the C compiler. Section 6 details the processor elements, before some conclusions (with a pointer to further information) in Section 7.

1. Processor Design – How to Start

1.1 Processor First, Memory Second

It is usual to build processors by concentrating on the processor first and then build a memory system with high performance caches to feed the processor bandwidth needs. In most computers, which means in most PCs, the external memory is almost treated as a necessary evil, a way to make the internal cache look bigger than it really is. The result is that the processor imposes strong locality of reference requirements onto the unsuspecting programmer at every step. Relatively few programs can be constructed with locality in mind at every step, but media codecs are one good example of specially tuned cache aware applications.

It is easy to observe what happens though when data locality is nonexistent by posting continuous random memory access patterns across the entire DRAM. The performance of a 2GHz Athlon (XP2400) with 1GByte of DDR DRAM can be reduced to about 300ns per memory access even though the DRAMs are much faster than that. The Athlon typically includes Translation Look-aside Buffers (TLBs) with 512 ways for both instruction and data references, with L1 cache of 16 Kbytes and L2 cache of 256 Kbytes or more.

While instruction fetch accesses can exhibit extremely good locality, data accesses for large scale linked lists, trees, and hash tables do not. A hash table value insertion might take 20 cycles in a hand cycle count of the source code but actually measures 1000 cycles in real time.

Most data memory accesses do not involve very much computation per access. To produce good processor performance, it is necessary, when using low cost high latency DRAM, to use complex multilevel cache hierarchies with TLBs hiding a multi level page table system with Operating System (OS) intervention occurring on TLB and page misses.

1.2 DRAM, a Short History

The Inmos 1986 Data book [7] first described the T414 Transputer alongside the SRAM and DRAM product lines. The data book describes the first Inmos CMOS IMS2800 DRAM. Minimum random access time and full cycle time was 60ns and 120ns respectively for 256Kbits. At the same time the T414 also cycled between 50ns to 80ns, they were almost matched. Today almost 20 years later the fastest DDR DRAM cycles about twice as fast with far greater I/O bandwidth and is now a clocked synchronous design storing 1Gbit. Twenty years of Moore's law was used to quadruple the memory size at a regular pace (about every 3 years) but cycle performance only improved slightly. The reasons for this are well known, but were driven by the requirement for low cost packaging. Since the first 4K bit 4027 DRAM from Mostek, the DRAM has used a multiplexed address bus, which means multiple sequence operations at the system and PCB level. This severely limits the opportunities for system improvement. Around the mid 1990s, IBM [8] and then Mosys [9] described high performance DRAMs with cycle times close to 5ns. These have been used in L3 cache and embedded in many Application Specific Integrated Circuits (ASICs).

In 2001 Micron and Infineon announced the 256Mbit Reduced Latency DRAM (RLDRAM) for the Network Processor Unit (NPU) industry, targeted at processing network packets. This not only reduced the minimum cycle time from 60ns to a maximum cycle time of 20ns, it threw the address multiplexing out in favor of an SRAM access structure. It further pipelined the design so that new accesses could start every 2.5ns on 8 independent memory banks. This product has generated little interest in the computer industry because of the focus on cache based single threaded processor design and continued use of low cost generic DRAMs.

1.3 Memory First, Processor Second

In the reverse model, the number of independent uncorrelated accesses into the largest possible affordable memory structure is maximized and then given enough computing resources to make the system work. Clearly this is not a single threaded model but requires many threads, and these must be communicating and highly interleaved. Here, the term *processes* are used in the occam sense and *threads* are used in the hardware sense to carry processes while they run on a processor. This model can be arbitrarily scaled by replicating the whole memory processor model. Since the memory throughput limit is already reached, additional processors must share higher-order memory object space via communication links – the essence of a Transputer.

With today's generic DRAMs the maximum issue rate of true random accesses is somewhere between 40ns to 60ns rate which is not very impressive compared to the Athlon best case of 1ns L1 cache but is much better than the outside 300ns case. The typical SDRAM has 4 banks but is barely able to operate 1.5 banks concurrently. The multiplexing of address lines interferes with continuous scheduling.

With RLDRAMs the 60ns figure can be reduced to 3.3ns with an FPGA controller giving 6 clocks or 20ns latency which is less than the 25ns instruction elapsed microcycle period. An ASIC controller can issue every 2.5ns with 8 clocks latency. The next generation RLDRAM3 scales the clock 4/3 to 533MHz and the issue rate to just below 2ns with 15ns latency with 64Mbytes.

There may even be a move to much higher banking ratios requested by customers, any number of banks greater than 8 helps reduce collisions and push performance closer to the theoretical limit. The author suggests the banking should follow DRAM core arrays, which means 64K banks for 16Kbit core arrays, or at least many times the latency/command rate.

Rambus is now also describing XDR2 a hoped for successor to XDR DRAM with some threading but still long latency. Rambus designs high performance system interfaces and not DRAM cores – hence no latency reduction. Rambus designed the XDR memory interface for the new Playstation3 and Cell processor. There are other modern DRAMs such as Fast Cycle DRAM (FCDRAM) but these do not seem to be so intuitive in use. There are downsides, such as bank and hash address collisions that will waste some cycles and no DIMM modules can be used.

This model of computing though can also still work with any memory type, but with different levels of performance. It is also likely that a hierarchy of different memory types can be used, with FPGA Block RAM inner most, plus external SRAM or RLDRAM and then outermost SDRAM. This has yet to be studied, combining the benefits of RLDRAM with the lower cost and larger size of SDRAM; it would look like a 1Million-way TLB.

It isn't possible to compete with mainstream CPUs by building more of the same, but turn the table upside down by compiling sequential software into slower highly parallel hardware in combination with FPGA Transputing, and things get interesting.

1.4 Transputer Definition

In this paper, a *Transputer* is defined as a scalable processor that supports concurrency in the hardware, with support for processes, channels and links based on the occam model. Object creation and access protection have been added which protects processes and makes them easier to write and validate. When address overflows are detected, the processor can use callbacks to handle the fault or allow the process to be stopped or other action taken.

1.5 Transputing with New Technology

The revised architecture exploits FPGA and RLDRAM with Multi Threading, multiple PEs and Inverted Page MMU. Despite these changes, the parallel programming model is intended to be the same or better, but the changes do affect programming languages and compilers in the use of workspaces and objects. Without FPGAs the project could never have been implemented. Without multi threading and RLDRAM, these other changes could not have occurred and the FPGA performance would be much poorer.

1.6 Transputing at the Memory Level

Although this paper is presented as a Transputer design, it is also a foundation design that could support several different computing styles that includes multiple or single PEs to each MMU. The Transputer as an architecture exists mostly in the MMU design which is where most of the Transputing instructions take effect. Almost all instructions that define occam behaviour involve either selective process scheduling and or memory move through channels or links and all of this is inside the MMU. The PEs start the occam instructions and then wait for the MMU to do the job usually taking a few microcycles, always less than 20 microcycles. The PE thread may get swapped by the process opcodes as a result.

1.7 Architecture Elements

The PE and MMU architectures are both quite detailed and can be described separately. They can be independently designed, developed, debugged, modeled and even replaced by alternate architectures. Even the instruction set is just another variable.

The new processor is built from a collection of PEs and a shared MMU, adding more thread slots until the MMU memory bandwidth limit is reached. The PE to MMU ratio varies with the type of memory attached to the MMU and the accepted memory load. The ratio can be higher if PEs are allowed to wait their turn on memory requests. The number of Links is treated the same way, more Links demand more MMU throughput with less available for the PEs. A Link might be viewed as a small specialized communications PE or Link Element (LE) with a Physical I/O port of an unspecified type. Indeed a Transputer with no PEs but many LEs would make a router switch. Another type of attached cell would be a Coprocessor Element or CE, this might be an FPU or hardware synthesized design.

1.8 Designing for the FPGA

The new processor has been specifically targeted to FPGAs, which are much harder to design for because many limits are imposed. The benefit is that one or more Transputers can be embedded into a small FPGA device with room to spare for other hardware structures, and at potentially low cost nearing $1 per PE based on 1 Block RAM and about

500 LUTs. The MMU cost is expected to be several times a single PE depending on capabilities included.

Unfortunately the classic styles of CPU design – even RISC designs – transferred to FPGA do not produce great results, and designs such as the Xilinx MicroBlaze [10] and the Altera NIOS [11] hover in the 120-150 Mips zone. These represent the best that can be done with Single Threaded Architecture (STA) aided by vendor insight into their own FPGA strengths. The cache and paging model is expensive to implement too. An obvious limit is the 32-bit ripple add path, which gives a typical 6ns limit. The expert arithmetic circuit designer might suggest Carry select, Carry look ahead and other well known speed up techniques [12], but these introduce more problems than they solve in FPGA. Generally VLSI transistor level designs can use elaborate structures with ease; a 64 bit adder can be built in 10 levels of logic in 1ns or less. FPGAs by contrast must force the designer to use whatever repeated structure can be placed into each and every LUT cell – nothing more and nothing less. A 64-bit ripple adder will take 12ns or so. Using better logic techniques means using plain LUT logic, which adds lots of fanout and irregularity. A previous PE design tried several of them and they each consumed disproportionate amounts of resources in return for modest speed up over a ripple add. Instead, the best solution seems to be to pipeline the carry half way and use a 2-cycle design. This uses just over half the hardware at twice the clock frequency. Now 2 PEs can be had with a doubling of thread performance.

1.9 Threading

The real problem with FPGA processor design is the sequential combinatorial logic, the STA processor must make a number of decisions at each pipeline clock and these usually need to perform the architecture specified width addition in 1 clock along with detecting branch conditions and getting the next instruction ready just in time, difficult even in VLSI.

Threading has been known about since the 1950s and has been used in several early processors such as the Control Data Corp CDC 6600. The scheme used here is Fine Grained or Vertical Multi Threading which is also used by the Sun Niagara (SPARC ISA), Raza (MIPS ISA), and the embedded Ubicom products [13, 14, 15]. The last 2 have been focused towards network packet processing and wireless systems. The Niagara will upgrade the SPARC architecture for throughput computing in servers. A common thread between many of these is 8 PEs, each with 4-way threading sharing the classic MMU and cache model. The applications for R16 are open to all comers using FPGA or Transputer technology.

The immediate benefit of threading a processor is that it turns it into a DSP-like engine with decision making logic given N times as many cycles to determine big changes in flow. It also helps simplify the processor design; several forms of complexity are replaced by a more manageable form of thread complexity which revolves around a small counter. A downside to threading is that it significantly increases pressure on the traditional cache designs but in R16, it helps the MMU spread out references into the hashed address space.

Threading also lets the architect remove advanced STA techniques such as Super Scalar, Register Renaming, Out-of-Order Execution and Very Long Instruction Word (VLIW) because they are irrelevant to MTA. The goal is not to maximise the PE performance at all cost, instead it is to obtain maximum performance for a given logic budget, since more PEs can be added to make it up. More PE performance simply means less PEs can be attached to the MMU for the same overall throughput: the MMU memory bandwidth is the final limit.

1.10 Algorithms and Locality of Reference, Big O or Big Oh

Since D. Knuth first published *'The Art of Computer Programming'* Volumes 1-3 [16] from 1962, these tomes have been regarded as a bedrock of algorithms. The texts describe many algorithms and data structures using a quaint MIX machine to run them with the results measured and analyzed to give big O notation expressions for the cost function. This was fine for many years while processors executed instructions in the same ballpark or so as the memory cycle time. Many of these structures are linked list or hashing type structures and do not exhibit much locality when spread across large memory, so the value of big O must be questioned. One of the most important ideas in computing: random numbers can not be used in indexing or addresses without paying the locality tax except on very small problems.

1.11 Pentium Grows Up

When the 486 and then Pentium-100 were released, a number of issues regarding the x86 architecture were cleaned up: the address space went to a flat 32-bit space, segments were orphaned and a good selection of RISC-like instructions became 1-cycle codes. The Pentium offered a dual data path presenting even more hand optimization possibilities. This change came with several soft cover optimization texts by authors such as M. Abrash [17], and later M. Schmit [18], and R. Booth [19] that concentrated on making some of the heavier material in Knuth and Sedgewick [20] usable in the x86 context. At this time the processors clocked near 100MHz and were still only an order faster than the DRAM and caches were much smaller than today. The authors demonstrated assembly coding techniques to hand optimize for all aspects of the processor as they understood it. By the time the Out-of-Order Pentium Pro arrived, the cycle counting game came to an end. Today we don't see these texts any more; there are too many variables in the architecture between Intel, AMD and others to keep up. Few programmers would want to optimize for 10 or more processor variations some of which might have opposing benefits. Of course these are all STA designs.

Today there is probably only one effective rule: memory operations that miss the cache are hugely expensive and even more so as the miss reaches the next cache level and TLBs. But all register-to-register operations and even quite a few short branches are more or less free. In practice the processor architects took over the responsibility of optimizing the code actually executed by the core by throwing enough hardware at the problem to keep the IPC from free falling as the cache misses would go up. It is now recognized by many that as the processor frequency goes up the usual trick of pushing the cache size up with it doesn't work anymore since the predominant area of the chip is cache which leaks. Ironically DRAM cells (which require continued refreshing) leak orders of magnitude less than SRAM cells: now if only they could just cycle faster (and with latency hiding, they effectively can).

That does make measuring the effectiveness of big O notation somewhat questionable if many of the measured operations are hundreds of times more expensive than others. The current regime of extreme forced locality must force software developers to either change their approach and use more localized algorithms or ignore it. Further most software running on most computers is old, possibly predating many processor generations, the operating system particularly so. While such software might occasionally get recompiled with a newer compiler version, most of the source code and data structures were likely written with the 486 in mind rather than the Athlon or P4. In many instances, the programmers are becoming so isolated from the processor that they cannot do anything

about locality … consider that Java and .NET use interpreted bytecodes with garbage collecting memory management and many layers of software in the standard APIs.

In the R16, the PEs are reminiscent of the earlier processors when instructions cycled at DRAM speeds. Very few special optimizations are needed to schedule instructions other than common sense general cases making big O usable again. With a cycle accurate model, the precise execution of an algorithm can be seen; program cycles can also be estimated by hand quite easily from measured or traced branch and memory patterns.

2. Building a New Transputer in 8 Steps

Acronyms: Single-Threaded Architecture (STA), Multi-Threaded Architecture (MTA), Virtual Address (VA), Physical Address (PA), Processor Element (PE), Link Element (LE), Co-processor Element (CE).

[1] Change STA CPU to MTA CPU.

[2] Change STA Memory to MTA Memory.

[3] Hash VA to PA to spread PA over all banks equally.

[4] Reduce page size to something useful like a 32-byte object.

[5] Hash object reference (or handle) with object linear addresses for each line.

[6] Use objects to build processes, channels, trees, link lists, hash tables, queues.

[7] Use lots of PEs with each MMU, also add custom LEs, CEs.

[8] Use lots of Transputers.

In step 1, the single-threaded model is replaced by the multi-threaded model; this removes considerable amounts of design complexity in hazard detection and forwarding logic at the cost of threading complexity and thread pressure on the cache model.

In step 2, the entire data cache hierarchy and address-multiplexed DRAM is replaced by MTA DRAM or RLDRAM which is up to 20 times faster than SDRAM.

In step 3, the Virtual to Physical address translation model is replaced by a hash function that spreads linear addresses to completely uncorrelated address patterns so that all address lines have equal chance to be different. This allows any lg (N) address bits to be used for the bank select for N-way banked DRAM with the least amount of undesired collisions. This scheme is related to Inverted Page Table MMU where the tables point to conventional DRAM pages of 4 Kbyte or much larger and use chained lists rather than rehashing.

In step 4, reduce the page size to something the programmer might actually allocate for the tiniest useful object, a string of 32 bytes or a link list atom or a hash table entry. This 32-byte line is also convenient for use as the burst block transfer unit which improves DRAM efficiency using DDR to fetch 4 sequential 64-bit words in 2 clocks which is 1 microcycle. At this level, only the Load Store operations use the bottom 5 address lines to select parts of the lines, otherwise the entire line is transferred to ICache, or to and from RCache, and possibly to and from outer levels of MTA SDRAM.

In step 5, objects are added by use of a private key or handle or reference into the hash calculation. This is simply a unique random number assigned to the object reference when the object is created by `New[]` using a Pseudo-random number generator (PRNG) combined with some software management. The reference must be returned to `Delete[]` to reverse the allocation steps. The price paid is that every 32-byte line will require a hit flag to be set or cleared. Allocation can be deferred until the line is needed.

In step 6, the process, channel, and scheduler objects are created that use the basic storage object. At this point the MMU has minimal knowledge of these structures but has some access to a descriptor just below index 0, and this completes a basic Transputer. Other application level objects might use a thin STL like wrapper. Even the Transputer occam support might now be in firmware running on a dedicated PE or thread but perhaps customized to do the job of link list editing schedule lists.

In step 7, combine multiple PEs with each MMU to boost throughput until bandwidth is stretched. Mix and match PEs with other CEs and LEs to build an interesting processor. A CE could be a computing element like an FPU from QinetiQ [21] or a co-processor designed in occam or Verilog that might run as software or then switched to a hardware design. A LE is some type of Link Element, Ethernet port etc. All elements share the physical memory system but all use private protected objects, which may be privately shared through the programming model.

In step 8, combine lots of Transputers first inside the FPGA, then outside to further boost performance using custom links and the occam framework. But also remember that FPGAs have the best value for money with the middle size parts and also the slower grades. While the largest FPGA may hold more than 500 Block RAMs, they are limited to 250 PEs before including MMUs and likely would be starved for I/O pins for each Transputer MMU to memory port. Every FPGA has a limit on the number of practical memory interfaces that can be hosted, because each needs specialized clock resources for high speed signal alignment.

Some systolic applications may be possible with no external memory for the MMU, instead using spare local Block RAMs. In these cases, many Transputers might be buried in an FPGA if the heat output can be managed. Peripheral Transputers might then manage external memory systems. The lack of internal access to external memory might be made up for by use of more Link bandwidth using wider connections.

3. Summary of Current Status

3.1 An FPGA Transputer Soft Core

A new implementation of a 32-bit Transputer is under development targeted for design in FPGA at about 300MHz, but also suitable for ASIC design at around 1GHz. Compared to the last production Transputers, the new design is probably 10 to 40 times faster per PE in FPGA, and can be built as a soft core for several dollars worth of FPGA resources and much less in an ASIC ignoring the larger NRE issue.

3.2 Instruction Set

The basic instruction word format is 16 bits with an optional 3 more 16 bit prefixes. The instruction set is very simple using only 2 formats. The primary 3 register RRR form and the 1 register with literal RL form. The prefix can be RRR or RL and follows the meaning of the final opcode. Prefixes can only extend the R and L fields. The first prefix has no cycle penalty so most instructions with 0 or 1 prefix take 1 microcycle.

The R register specifier can select 8, 64, 512, or 4096 registers mapped onto the process workspace (using 0-3 prefixes). The register specifier is an unsigned offset from the frame pointer (`fp`). The lower 64 registers offset from `fp` are transparently cached to the register cache to speed up most RRR or RL opcodes to 1 microcycle. Register references above 64 are accessed from the workspace memory using hidden load store cycles. Aliasing between the registers in memory and register cache is handled by the hardware. From the compiler and programmer's point of view, registers only exist in the workspace memory and the processor is a memory-to-memory design. By default, pointers can reach anywhere in the workspace (`wp`) data side and, with another object handle, anywhere through other objects. Objects or workspace pointers are not really pointers in the usual sense, but the term is used to keep familiarity with the original Transputer term. For most opcodes, `wp` is used implicitly as a workspace base by composing or hashing with a linear address calculation.

Branches take respectively 1, 2, or several microcycles if not taken, taken near, or taken far outside the instruction cache. Load and Store will likely take 2 microcycles. Other system instructions may take longer. The instructions conform to recent RISC ISA thinking by supplying components rather than solutions. These can be assembled into a broad range of program constructs.

Only a few very simple hand prepared programs have been run so far on the simulators while the C compiler is readied. These include Euclid's GCD and a dynamic branch test program. The basic branch control and basic math codes have been fully tested on the pipeline model shown in the schematic. The MMU and the Load Store instructions are further tested in the compiler. Load and Store instructions can read or write 1, 2, 4 or 8 byte operands usually signed, and the architecture could be upgraded to 64 bits width. For now registers may be paired for 64-bit use.

Register R0 is treated as a read 0 unless just written. The value is cleared as soon as it is read or a branch instruction follows (taken or not). Since the RRR codes have no literal format, the compiler puts literals into RRR instructions using a previous load literal signed or unsigned into R0. Other instructions may also write R0, useful for single use reads.

3.3 Multi Threaded Pipeline

The PEs are 4-way threaded and use 2 cycles (a microcycle) to remove the usual hazard detection and forwarding logic. The 2-cycle design dramatically simplifies and lowers the FPGA cost to around 500 LUTs from a baseline of around 1000 LUTs, and 1 or more Block RAMs of 2 KBytes per PE giving up to 150 Mips per PE. The total pipeline is around 15 stages, which is long compared to the 4 or 5 stages of MIPS or ARM processors; but instructions from each of the 4 threads use only every fourth pair of pipelines. The early pipeline stage includes the instruction counter and ICache address plus branch decision logic. The middle pipeline stage is the instruction prefetch expansion and basic decode and control logic. The last stage is the datapath and condition code logic. The PEs execute code until an occam process instruction is executed or a time limit is reached and then swap threads between the processes. The PEs have reached place and route in Xilinx FPGAs and the PE schematic diagram is included – see Figure 3.

3.4 Memory System

The MMU supports each different external memory type with a specific controller; the primary controllers are for RLDRAM, SRAM and DRAM. The memory is assumed to have a large flat address space with constant access time and is multi banked and low cost. All large transfers occur in multiples of 32-byte lines.

A single 32 MByte RLDRAM and its controller has enough throughput to support many PEs possibly up to 20 if some wait states are accepted. Bank collisions are mostly avoided by the MMU hashing policy. There are several Virtex-II-Pro boards with RLDRAM on board which can clock the RLDRAM at 300MHz with DDR I/O, well below the 400MHz specification, but the access latency is still 20ns or 8 clocks. This reduction loses 25% on issue rate but helps reduce collisions. The address bus is not multiplexed but the data bus may be common or split. The engineering of a custom RLDRAM FPGA PCB is particularly challenging, but is the eventual goal for a TRAM like module.

An SRAM and its very simple controller can support several PEs, but size and cost is not good. Many FPGA evaluation boards include 1MByte or so of 10 ns SRAM and no DRAM. The 8-way banking RLDRAM will be initially modelled with an SRAM with artificial banking on a low cost Spartan3 system.

An SDRAM or DDR DRAM and controller may only support 1 or 2 PEs and has much longer latency, but allows large memory size and low cost. The common SDRAM or DDR DRAM is burdened with the multiplexed Row and Column address that does not like true random accesses contrary to RLDRAM. These have effectively 20 times less throughput with at least 3 times the latency and severe limits on bank concurrency. But a 2 level system using either SRAM or RLDRAM with very large SDRAM may be practical.

For a really fast expensive processor, an all Block RAM design may be used for main memory. This would allow many PEs to be serviced with a much higher banking ratio than even RLDRAM and an effective latency of 1 cycle. The speed is largely wasted since all PEs send all memory requests through 1 MMU hash translation unit but the engineering is straightforward. An external 1MByte SRAM is almost as effective.

3.5 Memory Management Unit

The MMU exists as a small software library used extensively by the C compiler. It has not yet been used much by either of the simulators. The MMU hardware design is in planning. It consists of several conventional memory interfaces specific to the memory type used combined with the DMA engines, the hashing function, and interfaces for several PEs with priority or polled arbitration logic. It will also contain the Link layer shared component to support multiple Links or LEs.

3.6 Hash Function

The address hash function must produce a good spread even for small linear addresses on the same object reference. This is achieved by XORing several components. The MMU sends the bottom 5 bits of the virtual address directly to the memory controller. The remaining address is XORed with itself backwards and with shifted versions of the address and also the object reference and with a small table of 16 random words indexed by the lowest 4 address lines being hashed. The resulting physical line address is used to drive the memory above the 5 lower address bits. If a collision should occur, the hash tries again by including an additional counter value in the calculation. The required resources are mostly XOR gates and a small wide random table. For a multi-level DRAM system there may be a secondary hash path to produce a wider physical hashed address for the second memory.

3.7 Hit Counter Table

Of course in a classic hash system, eventually there are collisions, which require that all references be checked by inspecting a tag. For every 32-byte line, there is a required tag which should hold the virtual address pair of object reference and index. To speed things

up, there is a 2-bit hit counter for each line which counts the number of times an allocation occurred at the line. The values are 0, 1, many or unknown. This is stored in a fast SRAM in the MMU. When an access is performed, this hit table is checked and data is fetched anyway. If the hit table returns a 0, the access is invalid and the provided object reference determines the next action. If the hit-tables return a 1, the access is done and no tag needs to be checked. Otherwise the tag must be checked and a rehash performed, possibly many times. When a sparse structure is accessed with unallocated lines and the access test does not know in advance if the line is present, the tag must be checked.

3.8 The Instruction Cache

The Instruction Cache or ICache is really an instruction look-ahead queue. It can be up to 128 opcodes long and is a single continuous code window surrounding the instruction pointer (`ip`). When a process swap, function call, function return, or long branch occurs, the ICache is invalidated. For several microcycles the thread is stalled while the MMU performs 2 bursts of 32-byte fetch of opcodes (16 opcodes each) into the ICache into 2 of 8 available lines.

As soon as the second line starts to fill, `ip` may resume fetching instructions until another long branch occurs. When `ip` moves, it may branch backwards within the ICache queue for inner loops or branch forward into ICache. There will be a hint opcode to suggest that the system fetch farther ahead than 2 lines. If a loop can fit into the ICache and has complex branching that can jump forwards by 16 or more it should hint first and load the entire loop. The cycle accurate simulations show that the branch instruction mechanism works well, it is expected that half the forward branches will be taken and 1 quarter of those will be far enough to trigger a cache refill. The idea is to simply reduce the required instruction fetch bandwidth from main memory to a minimum.

While the common N-way set-associative ICache is considered a more obvious solution, this is really only true for STA processors, and these designs use considerably more FPGA resources than a queue design. The single Block RAM used for each PE gives each of the 4 threads an ICache and a Register Cache.

3.9 The Register Cache

The Register Cache (RCache) uses a single continuous register window that stays just ahead of the frame pointer (`fp`). In other words the hardware is very similar to the ICache hardware except that `fp` is adjusted by the function entry and exit codes, and this triggers the RCache to update. Similarly process swaps will also move `fp` and cause the RCache to swap all content. A fixed register model has been in use in the cycle simulation since the PE model was written. That has not yet been upgraded with a version of the ICache update logic since the fp model has not been completed either. Some light processes will want to limit RCache size to allow much faster process swaps, possibly even 8 registers will work.

3.10 The Data Cache

There is no data cache since the architecture revolves around RLDRAM and its very good threading or banked latency performance to hide multiple fetch latencies. However each RLDRAM is a 32 MByte chip and could itself be a data and instruction cache to another level of SDRAM. This has yet to be explored. Also a Block RAM array might be a lower level cache to RLDRAM but about 1000 times more expensive per byte and not much faster. It is anticipated that the memory model will allow any line of data to be exclusively in RCache, ICache, RLDRAM and so on out to SDRAM. Each memory system duplicates

the memory mapping system. The RLDRAM MMU layer hashes and searches to its virtual address width. If the RLDRAM fails, the system retries with a bigger hash to the slower DRAM and if it succeeds transfers multiple 32-byte lines closer to the core either to RLDRAM or either RCache or DCache but then invalidates the outer copy.

3.11 Objects and Descriptors

Objects of all sorts can be constructed by using the `New[]` *opcode* to allocate a new object reference. All active objects must have a unique random reference identifier usually given by a PRNG. The reference could be any width determined by how many objects an MMU system might want to have in use. A single RLDRAM of 32 MBytes could support 1 million unique 32-byte objects with no descriptor. An object with a descriptor requires at least 1 line of store just below the 0 address. Many interesting objects will use a descriptor containing multiple double links, possibly a call back function pointer, permissions, and other status information. A 32-bit object reference could support 4 billion objects, each could be up to 4 GBytes provided the physical DRAM can be constructed. There are limits to how many memory chips a system can drive so a 16 GByte system might use multiple DRAM controllers. One thing to consider is that PEs are cheap while memory systems are not.

When objects are deleted, the reference could be put back into a pool of unused references for reuse. Before doing this all lines allocated with that reference must be unallocated line by line. For larger object allocations of 1 MByte or so, possibly more than 32000 cycles will be needed to allocate or free all memory in one go, but then each line should be used several times, at least once to initialize. This is the price for object memory. It is perfectly reasonable to not allocate unless initializing so that uninitialised accesses can be caught as unallocated. A program might write a memory line with unknown by deallocating it; this sort of use must have tag checking turned on, which can be useful for debugging. For production, a software switch could disable that feature and could then avoid tag checking by testing the hit table for fully allocated structures. When an object is deleted, any dangling references to it will be caught as soon as they are accessed provided the reference has not been reused for a newer object.

3.12 Privileged Instructions

Every object reference can use a 32-bit linear space; the object reference will be combined with this to produce a physical address just wide enough to address the memory used. Usually an index that is combined with an object reference uses unsigned indexes and never touches the descriptor. But privileged instructions would be allowed to use signed indexes to reach in to the descriptors and change their contents. A really large descriptor might actually contain an executable binary image well below address 0. Clearly the operating system now gets very close to the hardware in a fairly straightforward way.

3.13 Back to Emulation

Indeed the entire Transputer kernel could be written in privileged microcode with a later effort to optimize slower parts into hardware. The STL could also be implemented as a thin wrapper over the underlying hardware. Given that PEs are cheap and memory systems are not, the Transputer kernel could be hosted on a privileged or dedicated or even customized PE rather than designing special hardware inside the MMU. If this kernel PE does not demand much instruction fetch bandwidth, then the bandwidth needed to edit the process and channel data structures may be the same, but the latency a little longer using software.

3.14 Processes and Channels

Whether the Transputer kernel is run as software on a PE or as hardware in the MMU, could also change the possible implementation of Processes and Channels. Assuming both models are in sync using the same data structures, it is known that process objects will need 3 sets of double linked lists for content, instance and schedule or event link stored in the descriptors for workspaces.

To support all linked list objects the PE or MMU must include some support for linked list management as software or hardware. In software that might be done with a small linked list package and executed as software with possible help from special instructions. As hardware the same package would be a model for how that hardware should work. Either way the linked list package will get worked out in the C compiler as the MMU has already done. The Compiler uses linked lists for the peephole optimizer and code emit, and could use them more extensively in the internal tree.

3.15 Process Scheduler

The schedule lists form a list of lists, the latter are for processes waiting for the same priority or the same point in future time. This allows occam style prioritized process to share time with hardware simulation. Every process instance is threaded through 1 of the priority lists.

3.16 Instruction Set Architecture Simulator

This simulator includes the MMU model so it could run some test functions when the compiler can finish up the immediate back end optimizations and encoding. So far it has only run hand written programs. This simulator is simply a forever switch block.

3.17 Register Transfer Level Simulator

Only the most important codes have been implemented in the C RTL simulator. The PE can perform basic ALU opcodes and conditional branch from the ICache across a 32-bit address space. The more elaborate branch-and-link is also implemented with some features turned off. The MMU is not included yet; the effective address currently goes to a simple data array.

3.18 C Compiler Development

A C compiler is under development that will later include occam [22] and a Verilog [23] subset. This is used to build test programs to debug processor logic; it will be self ported to R16. It currently can build small functions and compiles itself with much work remaining. The compiler reuses the MMU and linked list capabilities of the processor to build structures.

4. Instruction Set Architecture

4.1 Instruction Table

The R16 architecture can be implemented on 32- or 64-bit wide registers. This design uses a 32-bit register PE for the FPGA version using 2 cycles per instruction slot, but an ASIC version might be implemented in 1 cycle with more gates available. An instruction slot is

referred to as 1 microcycle. Registers can be paired for 64-bit use. Opcodes are 16 bits.Instruction Set Architecture

The Instruction Set is very simple and comes in 3 Register RRR or 1 Register with Literal RL format. The Register field is a multiple of 3 bits, the Literal field is a multiple of 8 bits. The PREFIX opcode can precede the main opcode up to 3 times so Register selects can be 3-12 bits wide and the Literal can be 1-4 bytes wide. The first PREFIX has no cycle penalty. These are used primarily to load a single use constant into an RRR opcode which has no literal field. The 3 Register fields are Rz <= Rx op Ry with a base value 0x0.z.x.y.

Table 1: Instruction Opcodes

[15:12]	0x000,008	0x080,088	0x800,808	0x880,888	RRR Format
0	add, adc	sub, sbb	and, orl	msk, xor	Arithmetic,Logicals
1	sll, srl	slc, src	slx, srx	srs, srr	Shifts Left Right
2	mul, div	rem, TBD	extends	swaps	Extended Math
3,4,5,6	TBD	TBD	TBD	TBD	Reserved
7	occam	object	memory	other	Reserved for MMU
8	ld1, ld2	ld4, ld8	st1, st2	st4, st8	Load, Store N Bytes
9	mv1, mv2	mv4, mv8	la1, la2	la4, la8	Block Move, Address
10	ji0, ji1	ceq, cne	cgt, cle	clt, cge	Conditional Reg Jump
11	jcc	teq, tne	tgt, tle	tlt, tge	Test Reg
[15:12]	**0xC0xx**	**0xC0xx**	**0xC8xx**	**0xC8xx**	**RL Format**
12	bcc	bcc	bcc	bcc	Conditional Branch
13	push	push	pop	pop	Arithmetic,Logicals
14	ldi	ldi	adi	adi	Load, Add Literal
15	cmi	cmi	PREFIX	PREFIX	Compare,FIX Literal

There are 16 basic arithmetic, logical and shift opcodes. Notes: `msk` is x&~y. In the shift set, `srs` and `srr` are shift right signed or rounded. The extended math opcodes are not defined but might include the usual `mul`, `div`, `rem` as well as various sign extends, byte swaps, bit count, priority encoding etc.

Blocks 3-6 are reserved or undefined. Block 7 is reserved for the MMU instructions to support occam, objects, and other memory operations.

Blocks 8 and 9 are related. Load and Store are opposing z <= x[y] or z => x[y]. Load address is obviously z <= &x[y]. Block Move or `mv` is x[] <= y[] for `Rz` transfers. In all 4 cases, the transfer size is specified as 1, 2, 4, 8 bytes, and all memory transfers are byte aligned. The MMU does *not* require special alignment rules because the design is much simpler than the usual cache & page table designs. DRAMs can be configured to stream bytes directly.

4.2 Calls, Returns, Jumps, Branches

Block 10 gives conditional `ji0`, `ji1`, `ceq`, `cne`, `cgt`, `cle`, `clt`, `cge` register defined jump opcodes. A set of 2 boolean *bit[0]* and 6 signed arithmetic tests on any `Rx` register can be used to conditionally call or jump through `Ry` with the option to save `ip` to `Rz`. Several variations can be found by using `R0` in any of the `Rz`, `Rx`, `Ry` fields:

`Rz` <= `ip`, if test (`Rx`, 0) `ip` <= `Ry` [+`ip`];

If `Rz` is `R0`, `ip` is not saved, used for calls and later the return, or even coroutine.

If `Rx` is `R0`, the test is *null*, the indirect version is selected, `and ip` is not added to `Ry`.

If `Ry` is `R0`, then the jump target is `R0+ip` or just `R0` if `Rx` is also `R0`. If `R0` was not previously just written the effect is either to jump to `0+ip` (a redundant skip) or `0` (a stop condition), otherwise the target is `R0+ip` or just `R0`. These 4 versions could be restated as *skip*, *stop*, *jmp relative* or *jmp absolute* to target using the `R0` value, all 4 still have the option to save `ip`. These would be used for *skip*, *stop*, *call*, *return*, and *switch* tables. Note that `R0` may have been written with a load literal for relative or absolute branching or any other normal instruction for computed branching.

4.3 Conditional `tcc` Test, `jcc` Jump Opcodes

Block 11 gives 6 *tcc* opcodes which are similar to the previous *ccc* opcodes except that `Ry` is used for the test with `Rx` with no jump with result saved in `Rz`:

`Rz` <= test (`Rx`, `Ry`); for signed arithmetic compares.

The jcc opcodes use `Rx` for the condition code and performs:

`Rz` <= `ip`, if (*ccc* is *T/F*) `ip` <= `ip` + `Ry`;

Again if `Rz` is `R0`, `ip` is not saved, and if `Ry` is `R0`, the result is a *skip* or *relative* or *computed relative conditional branch*. This is the long form of the *bcc* opcode.

4.4 Conditional `bcc` Branch Opcodes

Block 12 uses the literal compact form of relative conditional branch:

`Rz` <= `ip`, if (*ccc* is *T/F*) `ip` <= `ip` + L;

The 16 conditional *bcc* relative branches include an 8-bit signed literal offset, but have no option to save `ip`. If this is not flexible enough, the previous register jumps can be used. Clearly most short *if*, *for*, *while*, *goto*, *break*, *continue*, statements will use this *bcc* form or a prefixed version to increase the offset.

All the various branch and jumps above use the same underlying mechanism. The ip register is a 32-bit index, so an 8-bit offset can reach +127 to -128 opcodes into a code or program object cp. The MMU logically reads from `cp[0[ip]]`, the outer `[]` is the MMU hashing access function. The MMU may hold the various object `cp`, `wp`, `fp` registers.

4.5 Condition Codes

Modern RISC architects today frown on the classic condition code model although R16 supports both condition codes and register testing models. The issues regarding condition codes raised in the computer architecture texts are not relevant to R16 since it is not expected to ever implement superscalar, out of order, or register renaming, or VLIW techniques. Indeed the x86 and PPC use condition codes too, despite the architectural complexity that this adds.

The 8 condition flag values are C, V, N, Z, and Lt, Le, Ls, 0. The C, V, N, Z flags are decoded into the LessThan, LessEqual, LessSame flags for signed or unsigned tests, very similar to the 68000. The *bcc*, *jcc* opcodes take either the Rz or Rx field respectively as a 3 bit ccc field which selects 1 of the 8 condition flags above and this is conditionally inverted for the branch decision or test value. The mnemonics currently used are the same as the 68000 but arranged differently.

4.6 Registers

R16 does not define a limit on the number of registers available, `Rz`, `Rx`, `Ry` are simply offsets from `fp` into `wp` workspace memory. There is a process specified limit that forces register names below that to use the register cache (RCache). This value could be any multiple of 8 up to 64 or so. Register names beyond that limit force the PE to issue hidden Load, or Store microcycles to main memory in each particular case. This makes R16 both a register-to-register and memory-to-memory ISA. This is only practical because it uses no data cache and the minimum memory latency is actually less than the instruction latency. Typically 0 or 1 prefix is used, 0 for hot low registers and 1 for higher register access. There will be a mechanism for choosing other objects to index into besides `wp` for channels.

The RCache holds multiples of 8 registers, which is also the same size as 16 opcodes or 32 bytes, which is the MMU line size used for burst transfers around the processor and is also the basic memory allocation unit. Implementations can contain different amounts of RCache and ICache. In R16, each process thread is currently assigned up to 64 registers and 128 opcodes in the ICache (really a fetch-ahead queue). The compiler will work with function frames that are multiples of 8 registers bumping `fp` up or down by 8N on entry and exit. As `fp` moves, the RCache follows the `fp` window using almost the same hardware mechanism as when the ICache follows `ip`. Workspace accesses that alias into RCache are handled by having RCache drag a shadow hole over the workspace to trap these accesses.

4.7 Register R0 == 0

As in many RISC designs, `R0` is treated differently. `R0` value is usually read as 0 unless it was just written – but reverts back to 0 after reading or branching. This allows the 0 value to be very close and always available. No instructions in the RRR format have a literal field; instead the compiler should write `R0` with the literal using `ldi` `Rz`, L or `ldu` `Rz`, L for unsigned and then the following instruction consumes `R0`. If the literal is needed more than once in the immediate time line, it should not be stored in `R0`. Many ISAs with a literal option in each source field impose odd restrictions on the effective literal size and can have complex variable length encodings. Since RRR has no literal field, either `Rx`, or `Ry` can read `R0` to fetch a single use literal adding 1 microcycle for 8 or 16 bit values. Literals are most commonly needed by Load, Add, Compare, and conditional `bra` and the RL form allows these to do so with an 8 or 16 bit literal with no cycle penalty.

4.8 Functions

Function calls are simply branches or jumps with a save of ip to a designated safe place. The return instruction is just an indirect branch. The compiler places each stack frame starting from `fp[0]` and on up and much of this will overlap with the `RCache`. When a function is called, parameters are written further up `fp` starting 8n words higher just after last in scope variable. With 1 prefix almost all parameters will be register writes into `RCache`. On function entry, `fp` will be pulled up to the first parameter and the new function body gets to use the RRR opcodes for its own context. Since R fields cannot be negative, it can't see below `fp`, but it can use `fp` as a pointer into `wp[]`, load store can be used to explore the frames. To minimize the opcode sizes, the compiler may want to rearrange the frame so that the bottom few registers are used as hot temporaries with the parameters and locals just above this. Return results would likely be left in the hot registers, there wouldn't necessarily be any limit on the number of return values either.

4.9 Switch Statements

Whilst building switch statement target address, the skip opcode can save `ip` to `Rz` which can then be used to compute a target address to jump through. By placing this switch block code just after the last case or default statement, the switch jump can also save ip again just before jumping. Now the usual case breaks can be replaced by a return through that last saved `ip`.

There is an even more interesting version of the switch statement that takes advantage of the associative nature of the MMU memory system. In this scheme a sparse array of 32-byte labels is allocated as a label array object. The switch simply accesses the sparse array for the target address stored in the label cell. If a callback is stored in the label descriptor the switch code amounts to just a load target register sequence and register jump. The callback handles the default case where there is no valid label. The compiler would have to pre-build a label array for each switch statement. For a critical inner loop switch statement, only a few opcodes are needed for any arbitrary switch statement. There is also the possibility the label array can be dynamically altered, and since it is associative, many labels can switch to the same case allowing wild `xxx` values or case ranges something that is found in other languages such as Verilog. The C compiler will probably use this scheme.

4.10 Orthogonality

The use of prefixes allows the R16 ISA to give very compact code and is both orthogonal and very simple to decode. The various bra and jump codes are minor variations on the basic branch operation and mostly affect the `ICache` logic. While the math codes are variations on the add instruction that affect the data path, `RCache` and condition codes. The `Load`, `Store` and `Move` codes are also variations that affect mostly the MMU.

4.11 Hennessy & Patterson

On a side note the Hennessy & Patterson text [24] was thoroughly reviewed for typical statistics on all aspects of instruction opcodes. The text merely confirmed the obvious choices and designer prejudices. The highest priority was given to opcodes that the compiler must actually use to compile C codes and to support objects. Half of the RRR format is reserved for the future or application specific opcodes. Many well-known instruction sets were also referenced during the instruction selection. An unknown figure is

the Load Store to other opcode ratio, which will have some impact on the PE to MMU ratio.

5. Compiler Development

5.1 Compiler Introduction

This section describes the on-going C compiler development, which will later include occam and Verilog extensions. The compiler can now compile itself to about 8000 opcodes but requires much more work. Smaller examples such as *Quicksort* are almost ready to run, see Figure 1. The C compiler is based on a long run of earlier compiler work dating back to C to Verilog and Verilog to C translators and more recent C compiler prototypes. The Hanson and Fraser textbook, *"A Retargetable C Compiler: Design and Implementation"* [25], has been very useful. Half the project time has been used in compiler development.

5.2 Reusing MMU Components

The C compiler and the ISA and Cycle simulators are combined into a single project so that the MMU memory allocation code can also be used by the compiler to manage its own hash tables and linked list structures. Previously the compiler had its own memory management code but it became obvious that developing 2 similar packages did not make sense. If the compiler could use high-level functions that are in the R16 instruction set, then these opcodes would get thoroughly debugged in the compiler before even executing on the processor. Further when the compiler is later cross-targeted to R16, much of the code will be native instructions to the MMU to manage or access its memory rather than a compile of a software package. This will likely give the R16 native compiler a boost in performance compared to the PC hosted C compiler which runs the MMU as low locality functions.

5.3 Test Programs

To design and debug a processor requires that machine programs be prepared to run on several simulation models to prove correctness of the design as well as to explore various architectural ideas. R16 uses prefixed variable length opcodes, which quickly increases the effort of manual hand assembly of test programs more than a few lines long. A few test programs have been hand written in hex code and these have been enough to thoroughly check out the entire Instruction Cache and the conditional branch logic as well as the multi-threaded pipeline and data path. Some of these programs are dynamically changed to stress the processor hardware design while comparing the behaviour with a predicted model.

To push the development cycle much further requires that C functions be compiled to the ISA, which can then be compared against the same C code compiled into the simulator compiler package via the host Visual C compiler. This not only tests the processor design, it also tests the developing C compiler design. Differences in results between the ISA and cycle simulators running a cross compiled test program and the results of the same Visual C compiled C test program can usually identify bugs to be fixed.

The first batch of C test programs to be compiled and tested will be well-known examples such as Quicksort, and other classics from the Knuth or Sedgwick texts. The qualities desired of a test program are that it be moderately long, has a modest amount of input and output that can be simple array type problems with a variety of branching patterns. The Quicksort (Figure 1) can be used to stress test the memory system but its inner loops only use 4 opcodes (Figure 2) so it does not test much of the processor. Many

small programs will fit inside the ICache so the simulator can compare programs that generate much instruction fetch traffic to the MMU *versus* those that do not.

```
void quicksort(int a[], int l, int r) {
  int v,i,j,t;
  if (r>l) {  v = a[r]; i = l-1; j = r;
    while (true) {
      while (a[++i]<v);   // critical loop, should be 4 opcodes
      while (a[--j]>v);   // critical loop, should be 4 opcodes
      if (i>=j) break;
      t = a[i]; a[i] = a[j]; a[j] = t;
    }
    t = a[i]; a[i] = a[r]; a[r] = t;
    quicksort(a, l,  i-1);
    quicksort(a, i+1,  r);
  }
}
```

Figure 1: Quicksort in C

5.4 Native C Compiler

An immediate goal is to complete the C compiler so that it can retarget itself to the R16 and this is nearing completion even though much general work remains. When the C compiler is finally correct in compiling itself, R16 can run it to recompile itself to itself, and if it produces the same binary result, the project will have reached a major milestone. If it does not, then the C compiler will be instrumented to find trace differences between the systems.

5.5 Compiler Structure

The C compiler has previously used an Lcc lexer and parser solution with separate passes. It has since gone back to a single parse and lexical token scan on demand, which is much easier to add backtracking to within the source text. The parser directly emits an RPN tree, which is later scanned to emit a linked list of output codes, which then go through various stages of peephole optimization. In earlier compilers, a preprocessor was also built in and these early compilers used the Visual C preprocessor to build themselves. It was later realized that the compiler might be a lot cleaner if it did not use any preprocessor and that this would make the compiler much easier to self-compile. If the compiler can reach the point that inlined functions are supported, the quality will be much better in the source and in the final output. For the later Verilog support, inlining is required to be able to completely smash entire programs into a single module, so inlining is a major feature to implement. In a previous Verilog to C translator, this was already achieved.

5.6 V++ Compiler

Once the processor and C compiler are functional and reasonably correct, the long-term goal is to upgrade the current C compiler sources to compile the V++ language extensions, which are intended to include an occam and Verilog subset. Since the processor is designed to include occam support, the compiler must also include it. Further the processor will later generalize that to a Verilog event-driven subset, so that must be included too. This latter, however, is less well defined. The compiler framework has had these in mind all along. Note that Verilog and C have very similar expression syntax, but the Verilog block level syntax is more like Pascal. One could imagine occam as the primary language for writing parallel software and also use it to synthesize hardware blocks that can be placed on the same FPGA as a Transputer.

```
quicksort:           // missing fp +ve adjust
  tgt R1,r,l         // entry point
  brf L19            // if (r>l) {
  ldw R1,a,r         // REDUNDANT
  mov v,R1           // v = a[r];
  sub i,l,1          // i = l-1;
  mov j,r            // j = r;
  bra L23            // DEAD CODE
  bra L27            // DEAD BRA
L27:                 // while (true) {
  add i,i,1
  law R1,a,i         //   REDUNDANT
  ldw R0,R1,R0       //   ldw R0,a,i
  tlt R0,R0,v
  brt L27            //   while (a[++i]<v);
  bra L31            // DEAD BRA
L31:
  sub j,j,1
  law R1,a,j         //   REDUNDANT
  ldw R0,R1,R0       //   ldw R0,a,i
  tgt R0,R0,v
  brt L31            //   while (a[--j]>v);
  tge R1,i,j         //   if (i>=j) break;
  brf L36
  bra L26            // DEAD BRA
  bra L36            // DEAD BRA
L36:
 ldw R1,a,i          // REDUNDANT
  mov t,R1           // ldw t,a,i    t = a[i];
  law R1,a,i         // REDUNDANT
  ldw R2,a,j         // ldw R2,a,j
  stw R2,R1,R0       // stw R2,a,i   a[i] = a[j];
  law R1,a,j         // REDUNDANT
  stw t,R1,R0        // stw t,a,j    a[j] = t;
L23:
  tne R1,true,0      // DEAD CODE
  brt L27            // bra L27
L26:
  ldw R1,a,i
  mov t,R1
  law R1,a,i
  ldw R2,a,r
  stw R2,R1,R0
  law R1,a,r
  stw t,R1,R0
  mov T8,a
  mov T9,l
  sub T10,i,1
  bfn quicksort
  mov T8,a
  add T9,i,1
  mov T10,r
  bfn quicksort
  bra L19
L19:
  ret:         // missing fp -ve adjust
```

Figure 2: C compiler output for Quicksort

Figure 2 shows the output assembler emitted from the Quicksort function found in Sedgewick. The only edits applied to that output were to remove internal log columns to the left of the Label column and to add comments on where future work remains. Note that the labels have already been optimized away so that only those that are needed remain. The function does not yet have the stack frame instructions nor does it actually allocate variables to registers. The first redundant opcode shows `v = a[r]` which should be a single load opcode. The RPN internal structure evaluates the left and right side using pointer referencing and should merge those when the pointer is used once. The same applies to the `law-ldw` pairs, or *load-address,* then dereferences it. There are also several dead `bra` opcodes, these result from the *if-then-else* statements that have empty *else* parts. It was felt that rather than optimize that at the RPN stage, the peephole optimizer could perform that chore without caring why excessive `bra` codes are emitted.

5.7 Other Compilers

Compared to the original Transputer, new compilers will see a new instruction set that is very straightforward to compile to and very orthogonal. The biggest difference is the register Load, Store architecture and the opcodes are now variable length 16 bit opcodes with only two simple formats, RRR and RL. The MMU supports memory allocation and access checking in hardware on objects. That should be used at the language level otherwise entirely sequential programs can still be written using a single workspace, heap, stack.

The C support for `New[]` and `Delete[]` needs to be enhanced to allow the Transputer C programmer to direct object construction of full or partially allocated objects using hardware support. This could be done by allowing assembler level codes or calling a library of wrapper functions or by enhancing the C syntax.

6. Processor Element details

6.1 Multi Cycling 2 Clock Cycle Design

Most of the R16 instructions are executed in 1 microcycle (2 clocks), passing twice through a fast 16-bit data path. The operands are fetched and saved as 32-bit words. This requires 2 read and 1 write access to the register file in a dual port Block RAM; another access fetches a pair of 16 bit opcodes on a 32-bit aligned address. The operand reads occur on the odd clock and the write back and opcode fetch occur on the even clock. With 2 clocks, these 4 accesses can be handled by just 1 dual port Block RAM organized as 512 by 32 bits. This maximizes performance for minimum FPGA resources.

This leads to many simplifications that allow the entire PE to fit into about 500 LUTs and runs at 320MHz (after P/R) for about 150Mips or less on the the Virtex-II Pro. R16 achieves close to 0.45 IPC.

As IPC rises, the resources needed to obtain that performance rise much faster. CPUs that push IPC higher than 1 use orders of magnitude more resources for single-threaded designs and are fragile, especially if they use caches that miss too often with low locality code.

The R16 instruction decoder uses variable length prefixed opcodes up to 4 words total. This is simplified with the 2 clocks since each clock only considers 1 word; 2 clocks consider a single prefix opcode pair. Multiple prefixes are simply forced to take another microcycle since they are mostly used for infrequent long literals or long branches.

This design allows multiple PEs to be used with a shared MMU depending on the available memory bandwidth. With 2 PEs back to back, the throughput is clearly better than 1 PE running instructions every clock that uses more than twice the hardware resources. The 2-clock design is fundamental to circuit performance since it allows every PE to run at the maximum rate that a Block RAM or 16-bit adder or 3 levels of LUT logic for the entire design can achieve. Typically 3 levels of LUT logic equals 10 gate delays.

The design learned a great deal from R3, a previous, somewhat similar, ISA design which had a theoretical IPC of 1.3 and a data path clock rate of 300MHz. It eventually collapsed with 3 times the complexity and 1500 LUTs, and an actual IPC nearer to 1. Later the clock fell to 70MHz due to unplanned long control logic paths in the variable length instruction queue. While trying to rescue that design, it became obvious that 2 clocks could drastically simplify every part of the design and simplify the ISA too, so R16 was born. Many of the R3 best points survived in a more rational form.

6.2 Multi Processing N-way Design

This brings up the possibility that CPUs matched with several 32 MByte RLDRAMs using several PEs per MMU can achieve sustained zero locality memory accesses every O*3.3ns versus the Athlon worst case at 300ns regular DRAM cycle. That would give a theoretical 90/O speed-up for zero locality codes during this dead memory access period. The value for O averages between 1..3 due to collisions but with a possible unbounded limit that sometimes reaches 10 to 200 when the memory reaches 90% or more full. However, O also gradually increases when `New[]` and `Delete[]` are called in quick succession which dirties the hit table, but a periodic clean up scheme can lower O again. High locality data memory references are irrelevant with hash based MMU, except at the register level, but help the Athlon reach 1 ns cycles. The question is what percentage of memory accesses have no locality – maybe 1%?

6.3 Multi Threaded Architecture 4-way Design

The PE is divided nicely into 3 distinct blocks: the barrel controller, the instruction fetch queue plus prefix and opcode decode, and lastly the main data path. Each runs in permanent lock step in a never-ending cycle no matter what the 4 threads are doing within. Each block can be decomposed into smaller simpler sections, which can be analyzed or driven with test stimuli to verify each part independently.

Each pipeline has very little logic in any one stage. The longest critical path is either the 16-bit adder, the Block RAM cycle, or in many places 3 LUT levels worth of logic which is equivalent to 10 gates worth in an ASIC. This even varies somewhat by Xilinx FPGA family making it very hard to be optimal for all families. The relative simplicity allows the entire design to clock at the fastest rates for FPGAs, which is where only high performance DSP engines usually live. These 3 blocks have been through the Xilinx XST flow for Spartan3, Virtex-II Pro and now Virtex-4 with 225MHz, 320MHz and 320-370MHz results respectively after hand driven Place and Route. The Virtex-4 is too early to be taken further.

With 8 pipelines per microcycle, there is plenty of time to make some of the bigger decisions per instruction. In a classic single-threaded RISC, the same decision would have to be made entirely within 1 clock cycle which severely limits the rate of control decisions, often requiring stall states to keep up; this always lowers IPC.

6.4 Barrel Controller

The barrel controller controls the PE. It consists of 4 rows of thread state that rotates every odd clock in a circular fashion. The thread state contains a number of distinct fields for controlling the Instruction Queue, Instruction Cache, and the 4 ip values. The 4 rows include some combinatorial logic in a few places to adjust each field value to its next state.

The most complex part is the ip increment. In the barrel engine ip is only 7 bits wide, which eventually drives the ICache instruction pair fetch. The remainder of ip is left in the Block RAM register file. While instructions can be fetched from the ICache, ip follows the true position of the instruction pointer modulo 128. When branch opcodes occur, 1 microcycle is used to recompute the true `ip` for the next instruction and to leave that in some register. If the branch decision is true, a second microcycle performs the `ip` += offset or `ip` = target, depending on the instruction options. The branch decision logic includes the 8 to 1 condition code selection and various related logic. No presentable diagram available at this time.

6.5 Processor Schematic

Figure 3 shows both the data path and the instruction fetch queue and prefix path. Most prominent at top left is the Block RAM driven by microcode z, x, y addresses and the Thread Ids. A pair of operands flows straight down the 2-cycle data path and back to the Block RAM on the other cycle phase about 5 clocks later. This is matched by a 32-bit pair of opcodes sent to the IQ FIFO usually 2 by 2 as the IQ empties.

The other side of the IQ FIFO pulls opcodes out along with any prefix and arranges them into the 4 Microcode fields M3-M0 along with an optional delayed upper prefix pair. Only M0 drives control decodes. M3-M1 carry prefix payload which goes to the z, x, y address logic or literal input path. Not shown are the many small logic blocks that make this all work. This view accurately reflects the C Cycle RTL and Verilog RTL codes.

7. Conclusions

This paper reports a different way to obtain good performance from an FPGA processor by using a relatively simple processor design that can be clocked at DSP speeds. It can be replicated to increase performance and has enough memory bandwidth to allow several PEs to remain busy using a threaded DRAM rather than the usual multi level cache with paged table system. It is difficult to predict how performance will compare with other processors or even the past Transputer until it can be benchmarked, but early studies suggest that each PE may be only 5-8 times slower than a full Athlon XP2400 for a few dollars of FPGA resources (based on a huge *Quicksort* trace analysis).

Further it is hoped that bringing forth a new design for a Transputer will prove to have been the right thing to do; only time can tell. With continued development, most of this work should come to fruition.

This paper is an edited version of a longer (40+ pages) document that additionally provides: details of simulators for cycles and instructions, support for objects and processes, details on inter-processor links and FPU co-processors, support for high performance computing, ASIC implementation, FPGA tools and hardware/software co-design (with V++ Verilog, occam and C).

Future developments include completing the compiler, the PE and MMU designs, bringing up the FPGA board and later designing or finding a TRAM like FPGA RLDRAM

module with space for another function. If the project can be commercialized there is plenty of work for a dozen or even a hundred people.

Acknowledgement

The author gratefully acknowledges the reviewers' advice and comments on this paper, the result is a paper that became much more clearly focussed on the important aspects. The author also thanks Ruth Ivimey-Cook for preparing the processor illustration and for the initial paper that triggered the "what if, why not" thought. Also, thanks to the editors for making it possible to present the ideas here and for cleaning up the text and layout. Thanks especially to my wife for enduring many years of isolated development. I also thank the regular posters at the `comp.arch` newsgroup for many interesting computer architecture conversations; a lot can be learned by listening in. Finally, I thank Inmos for having existed and allowing me to spend my early years there.

References

[1] R. Ivimey-Cook, *Legacy of the Transputer*. In *'Architectures, Languages and Techniques for Concurrent Systems (WoTUG 22)'*, IOS Press, Amsterdam, 1999
[2] P. Walker, *Hardware for Transputing without Transputers*. In *'Parallel Processing Developments (WoTUG 19)'*, IOS Press, Amsterdam, 1996
[3] R. Meenakshisundaram, *ClassicCmp,* `http://www.classiccmp.org/`
[4] J. Gray, *FPGA CPU News,* `http://www.fpgacpu.org/`
[5] M. Tanaka et al, *Design of a Transputer Core and its Implementation in an FPGA,* CPA2004, IOS Press, Amsterdam, 2004
[6] Micron Technology, Inc., `http://www.micron.com`
[7] SGS-THOMSON Microelectronics, *Inmos Databook 1986*
[8] IBM United States, `http://www.ibm.com/`
[9] Monolithic Systems Technology, Inc., `http://www.mosys.com`
[10] Xilinx Inc., *MicroBlaze RISC Architecture*, `http://www.xilinx.com`
[11] Altera Inc., *NIOS RISC Architecture*, `http://www.altera.com`
[12] M. Flynn, S. Oberman, *Advanced Computer Arithmetic Design*, Wiley. ISBN 0-471-41209-0, 2001, 1-21.
[13] Sun Microsystems, *Niagara*, `http://www.sun.com`
[14] Raza Microelectonics Inc., `http://www.razamicroelectronics.com/products/xlr.html`
[15] Ubicom, `http://www.ubicom.com`
[16] D. Knuth, *The Art of Computer Programming*, Vols 1-3, Addison-Wesley. ISBN 0-201-89683-4, 1997.
[17] M. Abrash, *ZEN of Code Optimization*, Coriolis Group Books. ISBN 1-883577-03-9, 1994.
[18] M. Schmit, *Pentium Processor Optimization Tools*, AP Professional. ISBN 0-12-627230-1, 1995.
[19] R. Booth, *Inner Loops*, Addison Wesley. ISBN 0-201-47960-5, 1997.
[20] R. Sedgewick, *Algorithms in C*, p 118, Addison Wesley. ISBN 0-201-51425-7, 1990.
[21] QinetiQ, `http://www.qinetiq.com`
[22] A. Burns, *Programming in occam2*, Addison-Wesley. ISBN 0-201-17371-9, 1988.
[23] D.E. Thomas and P.R. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers. ISBN 1-4020-7089-6, June 2002, 1-23.
[24] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, pp 69-118, Elsevier Science & Technology Books. ISBN 1-55860-329-8, August 1995.
[25] C. Fraser and D.R. Hanson, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley. ISBN 0-805-31670-1, January 1995.

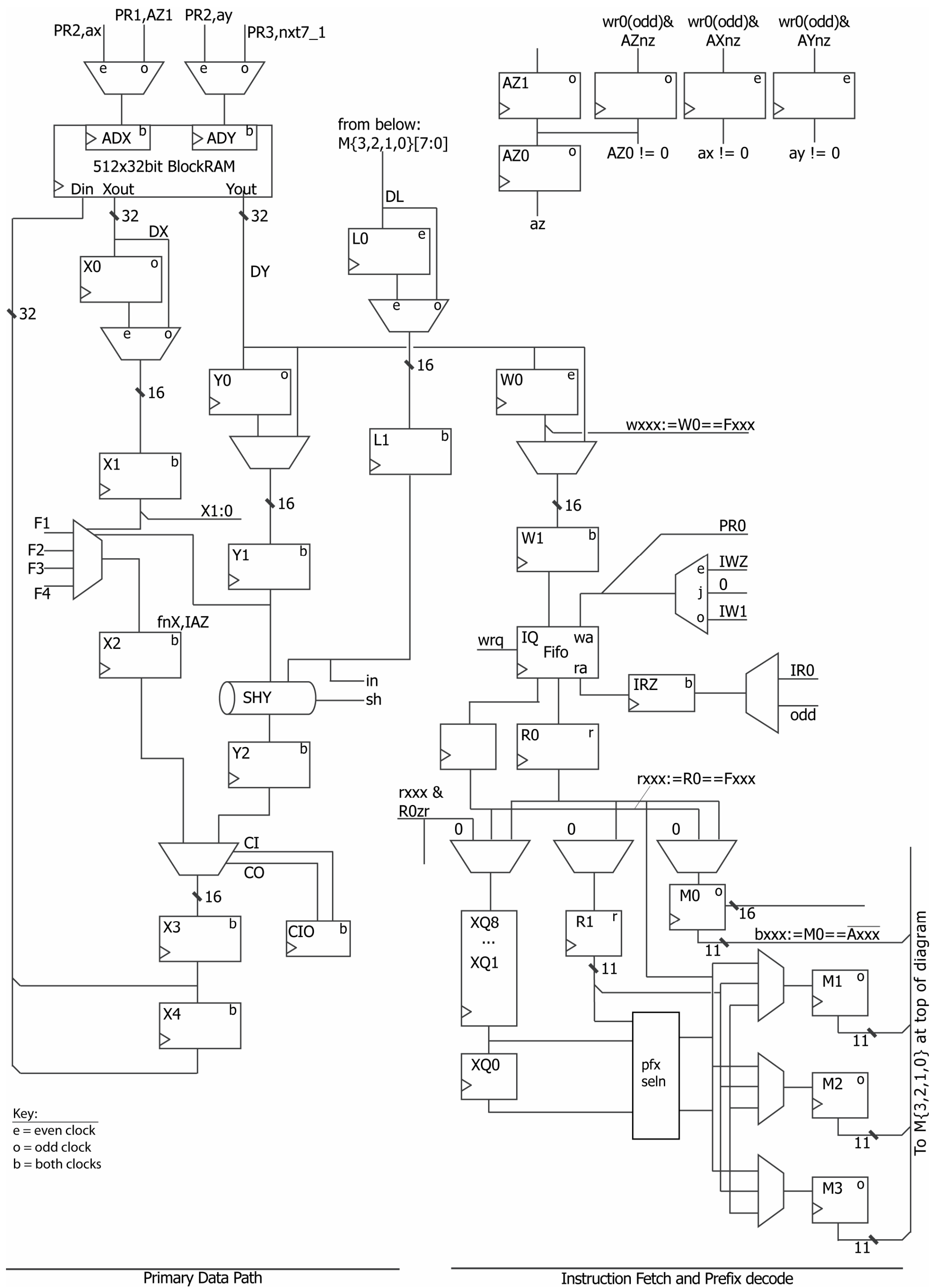

Figure 3: Schematic for the Data Path & Instruction Queue

Appendix: Main Terms Used Throughout the Paper

ALU	Arithmetic Logic Unit, centre of most data paths.
ASIC	Application Specific Integrated Circuit, typically 1-5 times faster than FPGA.
Callback	Function pointer stored in the descriptor called by the MMU when an illegal access has occurred, it might be harmless or serious. A zero field defaults the action to possibly stopping the process or raising an exception. Permissions used to determine if an object is executable, readable, writeable etc.
CE	Computing Element or co-processor designed by others that looks like a PE.
Channel	Used to synchronize communicating processes and transfer data either processor to processor or process to process respectively.
CISC	Complex Instruction Set Computer, typically memory to register.
CLA	Carry Look Ahead uses a tree of adders, often 4 way, adds in log time.
CPI	Cycles Per Instruction, inverse used on earlier microprogrammed CPUs.
CS	Computer Science.
CSA	Carry Select Adder, try 2 solutions with carry in of 0 and 1, select one later.
CSP	Communicating Sequential Processes, a way of looking at parallel programs as processes modelling hardware with channels and links replacing wires.
Cycle	Simulation sometimes used instead of RTL simulation but often in C or HDL
DCache	Data Cache, R16 uses an inverted page scheme in RLDRAM instead.
Descriptor	Indicates whether the object is workspace, channel, simple array, sparse etc.
DMA	Direct Memory Access minimal overhead block move, supports all vector operations, store moves, Links, Channels, process swaps, cache updates.
DSP	Digital Signal Processor, either special purpose CPU or special HW.
EDA	Electronic Design Automation or CAD specifically targeted to chip design. Used to be schematic based, with timing driven HDL simulation, now almost entirely Synthesis, STA and automatic P/R.
EE	Electronic Engineer.
FIFO	First In First Out.
FPGA	Field Programmable Gate Array, essentially reconfigurable HW.
FPU	Floating Point Unit, typically IEEE standard, a special purpose CE.
FW	Firmware as in bitfile or EPROM to configure an FPGA or embedded CPU
HandelC	C combined with occam used for higher level modelling and synthesis of hardware useful for converting algorithms to hardware with much less EE effort. A downside to HandelC and other commercial C based hardware synthesis is that they can not reach the entry level where free to use FPGA tools are available for Verilog and VHDL and are all proprietary. Others of note include PrecisionC, ImpulseC, SystemC, but these are intended more for system level design and co-design.
Handle	An object reference, basically a unique random no assigned by `New[]`.
HDL	Hardware Description Language, Verilog or VHDL (C, Pascal v ADA style)
HW	Hardware as in hard electronic stuff
ICache	Instruction Cache, R16 uses a simple instruction queue with hidden prefetch.
IP	Intellectual Property, especially for VLSI modules as soft cores
ip, wp, fp	Instruction, workspace and frame pointer, similar to original Transputer.
IPC	Instructions Per Cycle, up to one is fairly easy; above one becomes progressively harder and is what modern processors strive to achieve.
IPM	Inverted Page Mapping, an alternative IBM scheme used on higher end servers used to map a VA to PA by means of an address hash with many benefits in hardware reduction but some costs in variable access times. Usually used for large page sizes, in R16 it is used for 32-byte lines leading onto object memory management. IPM also requires additional tag and hit table RAM.
ISA	Instruction Set Architecture.
KRoC	Kent Retargetable occam Compiler, `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.
LE	Link Element which could be a DS Link, a USB or Ethernet or Video port.
Line	RCache, ICache and DMA transfers use 32-byte line by line burst transfers.
Link	Serial DMA interface with very low hardware costs connects to a Channel.
LUT	Look Up Table, basic FPGA component equivalent to a few or many logic gates
MAC	Multiplier Accumulator, multiplier plus wide summing adder in 1 circuit, used extensively by programmable DSPs and in FPGAs usually paired with a Block RAM. Virtex-4 FPGA devices build these into embedded DSP cores.
MMU	Memory Management Unit, usually paged tables, or inverted page tables. In R16 the MMU is the main hub for multiple PEs, LEs, CEs, but also delivers the Object support for Processes through a hash address scheme.

MTA	Multi-Threaded Architecture, recent trend to hide latency usually in CPUs.
NPU	Network Processor Unit used to process network packets.
NRE	Non Returnable Expense, design engineering costs.
Object	A block of memory with associated descriptor just below address 0, contains link list fields, callbacks, permissions, used by Transputer kernel and other processes.
occam	The original native language for the Transputer with small syntax that implemented CSP in a practical form, almost an assembly level language.
OoO	Out-of-Order, as in rearranging instructions in time order rather than broken.
OS	Operating System, usually software, but in Transputers partially in hardware.
P/R	Place & Route takes synthesized logic and completes final wiring etc., may include human Floor Planning to help tool produce better or worse results.
PA	Physical Address, the actual address that drives DRAM arrays.
PCB	Printed Circuit Board
PDL	Process Description Language, such as occam, or even Verilog or VHDL
PE	Processor Element, smaller incomplete CPU, requires MMU access.
PRNG	Pseudo-Random Number Generator, a special form of counter that counts through a random looking sequence. This is used to generate Object handles or references. Might be collected after de-allocation and reused.
Process	The basic unit of computing, models a hardware process or module. *Also* the other meaning describes semiconductor fabrication processes.
R16	This CPU, uses 16-bit 2-clock data path, follows previous R3 32-bit design.
RA	R16 Register Address, typically 3, 6 or even 9, 12 bit wide address, these are always mapped into VAs into Workspace memory at `fp[RA]` with hidden Load/Store operations for those beyond a specified limit but direct register access for those below. Such operations are both memory to memory and register to register. The RCache fills in a memory hole in the workspace.
RC	Reconfigurable Computing, changing FPGA function on the fly
RCache	Register Cache, R16 uses a sliding register window over the workspace.
RISC	Reduced Instruction Set Computer, typically Load/Store to register.
RPN	Reverse Polish Notation, used in compilers.
RTL	Register Transfer Level, construct compute pipeline codes in HDL or C
SOC	System-on-Chip, used by marketing more than EEs.
STA	Single-Threaded Architecture: most processors today. *Also* Static Timing Analysis, analysis of circuits saving untold simulations, largely replaced the edit and simulate with timing cycles. Combined with Synthesis, most designs are now modelled cycle accurate without detailed timing
STL	Standard Template Library, a C++ package of standard objects that could be easily mapped onto the object memory system.
SW	Software as in soft code stuff, although the definition can be blurred
Synthesis	HDL RTL code is synthesized into logic that can be Placed & Routed to the target device, it asks how fast, designer says 3ns, it tries its best.
Thread	Hardware threads in a PE execute a software process for a length of time.
TLB	Translation Look-aside Buffer, a small associative address cache.
TRAM	TRAnsputer Module, a Transputing PCB module about the size of a credit card.
Transputer	A processor that supports occam style processes and communications.
ULSI	VLSI * 10, no newer terms were added although SOC is in vogue today.
V++	A proposed language that combines HDL, C, occam (`PAR`, `SEQ`, `PAR+SEQ`) allowing different coding styles to be mixed or linked something like gcc, a common compiler framework with multiple front ends. This will permit algorithms to be developed in one language and then moved into another to reach other tools. Allows source code to run as executable code or synthesize as a co-processor possibly an LE connected to the MMU.
VA	Virtual address, the address constructed by Load/Store instruction. In most processors the VA to PA translation is performed by multilevel page table lookups accelerated with a bypass Address Cache or TLB but with relatively few entries easily broken with low locality or multi-threaded code.
VLSI	Very Large Scale Integration, full custom typically 5-25x faster than FPGA.

DDR	Double Data Rate, clocks data on both clock edges, typically 300 MHz clock
DIMM	Dual Inline Memory Module, the usual packaging format for DRAMs.
DLL	Delay Locked Loops, recent additions to FPGAs allow high speed serial I/Os. Note, all modern DRAM and many SRAM interfaces are now essentially communications systems that use extensive PLL, DLLs to align signal and clock edges. Some even perform line characteristics to tune the IO circuits.

PLL	Phase Locked Loops, used to multiply (and divide) clock frequencies
QDR	Quad Data Rate, clocks data at 4x the clock rate, requires fancy PLLs, DLLs.
WebPack	Xilinx version of free to use EDA FPGA package covering all modern but small to medium devices which can still include up to 1Million gate design equivalent functions. Altera has Quartus as near equivalent. Pay versions are unlocked for all the largest FPGAs and still only cost 1% of ASIC EDA tools. Both have neutral support for Verilog and VHDL.
XDR2	Newer version of XDR from Rambus that now supports threaded DRAM.
Block RAM	(Also BRAM) SRAM internal to FPGA, dual ported 18 Kbits in various shapes, Around 300 MHz cycle rate. Available in 4 to 550 or so instances, about $1 each best used for their bandwidth, not as bulk SRAM. Can be grouped for more bits.
DDRAM	Superseded SDRAM with the minor change of transferring data on both clock edges i.e. DDR.
DRAM	Dynamic RAM, hierarchical memory structure with varying access times depending on Bank, Column, Row, access times varying from 60ns to 1.5ns typically used RAS/CAS clocks and address and data multiplexing to save pins.
EEPROM	Electrically Erasable Programmable Read Only Memory, reusable but wears. Some FPGAs may include the EEPROM internally to boot the FPGA, high performance FPGAs generally boot from external ROM.
EPROM	Erasable Programmable Read Only Memory (using UV light, largely superseded by EEPROM).
FLASH	EEPROM packaged in small package such as low cost Smart Media used in digital cameras now usable for booting FPGAs with suitable boot interface. Historically EEPROM used specifically for FPGA has been very expensive. Note by definition all MOS devices wear out but EE devices much more so.
PROM	Programmable Read Only Memory, often 1 shot use only.
QDRAM	Expected to supersede DDRAM, QDR DRAM, even more bandwidth.
RLDRAM	Reduced Latency DRAM, very high performance DRAM, multi-banked with few bank restrictions, access times of 20ns, new command cycle every 2.5ns.
SDRAM	Synchronous DRAM superseded old style RAS/CAS DRAM and now uses 1 system clock and reduces RAS/CAS to mode control bits.
SRAM	Static RAM, memory structure with typical access time of 10ns or less
ARM	ARM processor highly successful but almost invisible embedded RISC processor. Notable features include good basic opcodes combined with optional shifts and conditional operations, highly protected IP licence.
HPC	High Performance Computing term used in scientific computing.
MIPS	Classic RISC processor also highly successful embedded processor that is also highly protected IP licence.
Niagara	A threaded processor design from Afara acquired by Sun Microsystems. It features 8 processor cores each 4 way threaded running at about 1GHz. Afara was also housed in a venture capital outfit connected to Atiq Raza.
PPC	Power PC architecture descended from the very early work of John Cocke's seminal RISC work. Of note is that is that it has many more instructions than other RISCs, it has 8 condition code sets selected by each opcode, it has been implemented in many forms and both 32 and 64 bits wide. In order and out of order, superscalar etc, the last standing high performance competitor to x86 at the high frequency end. Now featured as the processor of choice in the next generation of gaming systems in particular the Cell which some have likened to a Transputer because of the inclusion of 8 additional coprocessors.
Raza XLR	A threaded processor design using the MIPS ISA from Atiq Raza who was also the primary architect for the NexGen 686, and AMD Athlon series. Features 8 processor cores each 4 way threaded running at about 1.5 GHz, also supports RLDRAM aimed at network packet processing.
Tera MTA	A famous MTA architecture dating back to the Denelcor HEP designed by Burton Smith during the late 1980s and later reincarnated as Tera MTA. It featured multi threading in the processor and memory and could extract instruction level parallelism although at great hardware expense.
Ubicom	Another MTA 8 way threaded processor at 250 MHz for Wireless markets.