Beehive: A many-core computer for FPGAs (v5)

Chuck Thacker, MSR Silicon Valley

8 January, 2010

Introduction

This paper describes an experimental many-core computer implemented on a single FPGA. The goals of the design were to provide a system that could be easily modified for “What if?” experiments, and which could be made available to the academic community for research purposes without external IP entanglement. Cores available for research use are either opaque “black boxes” (Xilinx Micro Blaze), are expensive (ARM Cortex M1), or are large and complex (Sun and Leon SPARC cores). We wanted a core that is simple enough that it could be easily understood and modified, yet which is powerful enough that it can run large programs written in C or C#. We also wanted a system that could be built using only the basic Xilinx tools (ISE and ChipScope). There are better synthesis tools, but these are expensive and not available to many of our target users.

The system includes a number[1] of RISC cores, each with a local I/O system. The local I/O devices in each core include an RS232 controller, a multiply unit (implemented with a DSP) capable of doing a 32 X 32 2’s complement multiply in ten cycles, an inter-processor messaging facility for exchanging short messages between cores, and a locking unit to provide atomic operations.

Each processor core includes a 4KB direct-mapped, allocate-on-write, blocking data cache and a 4KB direct-mapped instruction cache. The caches are *not* coherent – coherence must be provided where needed through software, assisted by operations that allow a region of the data cache to be *flushed* or *invalidated*. It is possible to provide a larger cache with increased associativity, but this would come at the cost of more Block RAMs, which are the scarcest FPGA resource.

The processor is a fairly conventional 32-register RISC. It uses byte addressing for data accesses, and word addressing for instructions. Data and instruction accesses can address 8GB of DDR2 DRAM memory (the memory on one of the two DDR channels of the BEE3).

The processors are connected to each other and to the memory using a token ring interconnect. The interconnect carries a stream of 32-bit data items plus a 4-bit *SlotType* field and a 4-bit *SrcDest* field indicating the source or destination of an operation[2]. The ring addresses one of the most serious limitations of an FPGA many-core design: Routing congestion. Had we used a crossbar or hierarchical bus, routing a design with more than a few cores would be extremely problematic. With a ring, all inter-core wiring is local and relatively short, so it is possible to instantiate a large number of cores. The processor is small enough that we estimate that a single FPGA can have 16-32 cores, depending on the

[1] A parameter in module *RiscTop*.

[2] This field limits the number of cores to 15. It can be made wider if needed.

number and complexity of other high performance I/O devices. We plan to experiment with a wider and faster ring, but have not done this yet.

I/O devices such as the 1 GB and 10 GB Ethernet interfaces and the PCI Express endpoint provided by the BEE3 are implemented as separate nodes on the ring. Processors control these devices by exchanging messages with them, and the devices can do DMA directly to and from memory.

A separate version of the design, with a different memory controller[3], has been implemented for the Xilinx ML509 (XUPv5) development board. This board supports only 2GB of non-error-corrected memory, and the limited size of the LX110T FPGA used on the board limits the number of cores[4]. In addition, since ML509 board is populated with a lower speed-grade FPGA, the RISC cores run at 100 MHz, rather than 125 MHz as in the BEE3 version. This changes the implementation of the Ethernet controller slightly (since it *must* run at 125 MHz), but otherwise, the design is identical to the BEE3 version.

The CPU core

The data paths of the CPU are shown in Figure 1.

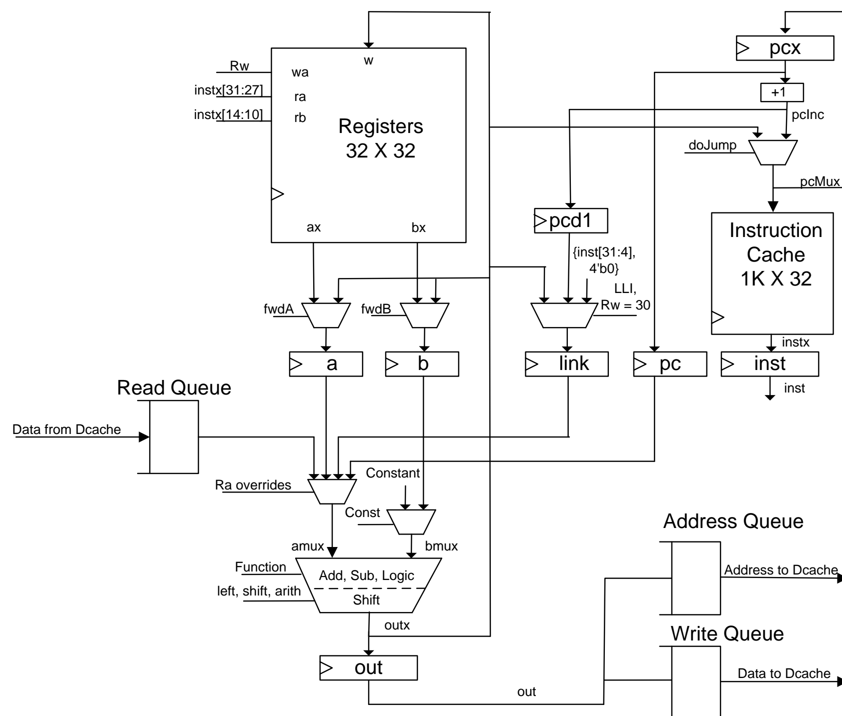

Figure 1: The simple RISC. This is a simplified schematic, since it doesn't show the details of the queues.

---

[3] This controller is an enhancement of the original BEE3 controller, implemented by Zhangxi Tan on U.C. Berkeley.

[4] I'm not sure what the limit is. The design presently contains 13 cores (since the Ethernet controller and the block copier each contain specialized cores). This exhausts the 4-bit SrcDest field, but the design uses only 50% of the LX110T's resources.

The primary data path consists of the register file, which contains 32 32-bit registers, and an ALU followed by a full barrel shifter capable of doing cyclic, logical, or arithmetic shifts.

All instructions have the same format, and consist of the fields shown in Figure 2:

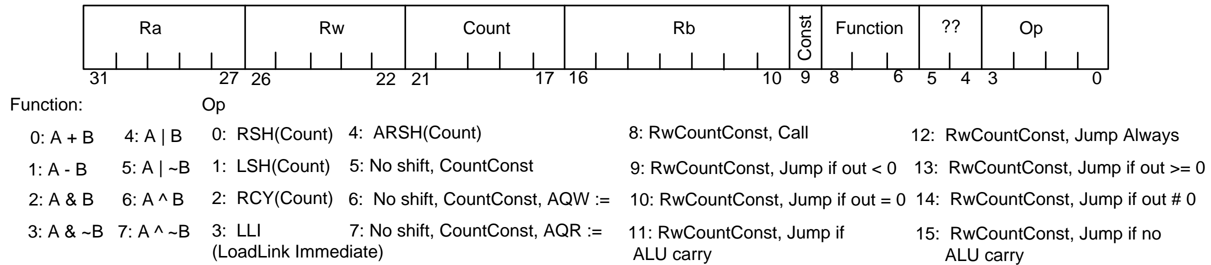

Figure 2: Instruction format. Bits labeled "??" are currently unused.

All instructions execute in three pipelined cycles. During the first cycle an instruction is fetched from the instruction memory and the register file is read into the a and b registers. If the previous instruction wrote a destination that is needed by the current instruction, the value (outx) is *forwarded* to a or b. During the second cycle, the a and b values are operated upon. At the end of the second cycle, the outx signals are latched at the Register w port, and are (usually) written into the register file during the third cycle.

The ALU functions are conventional, but some of the other encoding deserves comment:

1) Conditional jumps test the results of the *previous* instruction. There are actually 16 jumps, those shown in Figure 2, and 8 additional codes, only one of which (j7) is currently used by the debugging unit described later. The set of jumps used is selected by bit 26 of the jump instruction. Bit 26 = 0 selects the set shown in Figure 2, bit 26 = 1 selects the alternate set (j0 – j7). For the conditional jumps, if the condition is true, pcx and the shadow PC in the Instruction cache are written from outx, which contains the target address. Jumps (taken or not) suppress the write of RF, since the target of the jump is not very useful. This allows Rw[3:0] to be used in constant generation (see below). Note that the zero and negative jumps test the shifter output, while the carry test tests the result of the previous addition; it is undefined if the operation was not add or subtract. Taken jumps *nullify* the following instruction's execution[5].
2) Call is an unconditional jump that writes pcd1 (PC + 1) into the Link register. Both PC and the Link register are available as an amux output via *Ra overrides,* which use Ra to specify a register rather than a word in RF. This provides for PC-relative jumps. The link is also loaded from outx when Rw = 30.

[5] It is sometimes possible to fill the delay slot of a taken jump with a useful instruction, but if not, a nop must be used. Nullification eliminates the nops, favoring code space over execution speed.

3) If Const = 0, the low order five bits of Rb address the register file, and the resulting value is sent to the ALU via the bmux. If Const = 1, the Rb field is used as a constant. All seven bits of Rb are used in the generation of constants as described later.
4) The amux is used to select the PC, Link, or the *Read Queue* as the ALU input. Ra =31 selects PC, Ra = 30 selects the Link, and Ra = 29 selects the Read Queue. Other Ra values select RF [Ra].
5) The LLI instruction allows generation of a 28-bit constant. All state changes except the write into Link are suppressed, and Link [31:4] <= inst [31:4]. This allows a register to be loaded with a 32-bit constant in two instructions.

Memory Access

In conventional machines, a memory read specifies both the address in memory to be read and the register into which the returned data is to be placed. Similarly, a memory write specifies the source of the data (a register), and the memory location into which the data is to be written.

In this design[6], we decouple the address calculation from the source or destination selection. To issue a read, the output of the ALU/Shifter is sent to the Address Queue. A non-empty Address Queue causes the read to occur, and the resulting data is placed in the Read Queue. The output of this queue may be read by selecting Ra = 29. If the data has not yet returned from memory, the processor stalls.

Writes are handled similarly. Data is written to the memory when a write appears at the head of the Address Queue and the Write Queue is nonempty. The Write Queue may be written either before or after the Address Queue is written, since the write will not be started until both queues are nonempty[7].

Writes to the Address Queue are controlled by the Op field. If Op > 4, all shifting is suppressed, making Count available for other purposes. If Op = 6, the Address Queue is written from outx to start a read, and if Op = 7 the Address queue is written from outx to start a write. The Write Queue is written when Rw = 31.

By overloading the shifts to control writes to the address queue, we lose the ability to use the shifter in address generation, but this seems harmless.

Although the machine is basically word addressed, the compilers we wish to use require byte addressing. We do not provide full support for this, but it appears that the minimal support provided should be sufficient. A data address is treated as a 32-bit (word-aligned) byte address. The compilers always emit code in which the low order two address bits are 00. Compiled code never needs to access I/O space (such code will be written in assembler), so we use address bit 1 to select I/O space.

[6] This is not a new idea. It first appeared in this form in an architecture designed by W. Wulf in 1990. Unfortunately, his design was never built.
[7] This is not true for multiply or for writes to the Messenger. The operands must be placed in the write queue before writing the address queue.

For ease of implementation, and backwards compatibility with an earlier word-addressed design, all addresses beyond the address queue are *word* addresses. Whenever the address queue is loaded, bits 31:2 of the shifter output are loaded into AQ[29:0], AQ[31:30] are loaded with bits 1:0 of the shifter output. This makes it possible for assembly code to access the full 8GB. To access the high 4GB, an address with bit 0 = 1 is used. References to AQ below refer to the value after shifting.

The PC is a 31-bit word address, which is capable of addressing 8GB. This means that code can be located in a region of memory normally inaccessible to normal data accesses.

Constant Generation

By suppressing shifts, we make Count available for constant generation. Normally, Const = 0 selects the b output of RF as the input to the ALU. If Const = 1, a zero-extended constant is substituted for b. By default, this constant is Rb (7 bits). If Op > 4, {Count, Rb} is used instead. This provides a 12-bit constant in instructions that do not require shifts. In addition, a Jump instruction does not need Rw[3:0], since writing the target address of the jump to RF has little or no value. Instructions with Op >= 8 therefore have a 16-bit constant: {Rw[3:0], Count, Rb}.

Interrupts

Interrupts are TBD. So far, they have not been needed, and aren't implemented.

I/O devices and Coprocessors

All I/O is memory mapped. AQ[31] = 0 references memory space, AQ[31] = 1 references I/O space. There are only six local I/O devices on each CPU, so the device is selected by the low 3 bits of AQ. This leaves the remaining AQ bits free for other uses. The I/O devices (each of which is described in a Verilog module) are connected to the read, write, and address queues as shown in figure 3. Some of the devices also connect to external signals (e.g., the RS232) or to the ring interconnect network.
An I/O device is started when AQ is nonempty and it contains the address of the device. Devices can take many cycles to complete an operation. When the operation is complete, the device asserts a "done" signal that advances AQ.

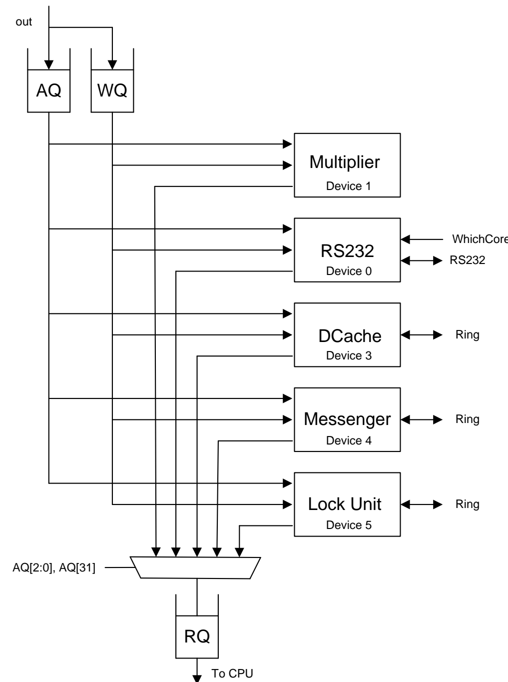

Figure 3: Local I/O devices. Device 2 (the output register) is not shown since it is simple enough that it doesn't use a separate module in the design.

The I/O devices are:

1) An RS232 interface for an RS232 line operating at 115,200 bps, with 8 data bits, one stop bit, no parity, and no flow control. The RS232 interface appears at I/O location 0. It consists of two sub-devices, the RS232 interface itself, and a 32-bit cycle counter that increments at the system clock rate. Reading location 0 with a[3] = 1 returns the cycle counter, a[3] = zero returns a word containing the following fields:

    Bits [7:0]: The received character if there is one,
    Bit 8: The "receive valid" flag indicating that bits 7:0 are valid,
    Bit 9: The "transmitter empty" flag which is true if it is OK to send a character,
    Bits 13:10: The number (1..nCores) of this core
    Bits 17:14: The number of the Ethernet controller's core, and
    Bits 24:18: The system clock speed in MHz. This is used indirectly to indicate the platform type: For BEE3, it is 125 MHz, for the ML509, it is 100 MHz.

    A write to location 0 takes a 10-bit data value in WQ. Bits 7:0 are the transmit character, bit 8 is the "receiver reset" bit, and bit 9 is the "load transmitter" bit. The RS232 is currently operated by polling. A shell program polls I/O location 0, waiting for a character to arrive (RQ[8] nonzero). When a character arrives, the shell responds by writing WQ [8] = 1, and AQ = 0. This allows the receiver to accept another character. To transmit a character, the shell reads I/O location 0, waiting for RQ [9] = 1. When it is true, the shell writes the character with WQ [9] = 1 to I/O location 0, which sends the character. There is no buffering, but humans can't type fast enough to overrun the receiver or the transmitter. Since there are several cores, the FPGA's single

RS232 connection must be multiplexed between the cores. This is controlled by the TC RISC processor in the memory controller, which is the default owner of the RS232 line. When the FPGA is configured, the TC initializes and calibrates the DRAM, then waits for user input. Typing a number followed by “s” switches the RS232 to the selected core. If the core doesn’t exist, or is not ready for RS232 input, you’re stuck. A core can return the RS232 to the TC by toggling bit 0 of output unit 2, which resets the selection. The current RISC shell does this when “r” is typed at the shell prompt.

2) A multiplier. Multiply is provided by a memory-mapped coprocessor at I/O location 1. The coprocessor uses an FPGA DSP unit to provide a 32 X 32 signed integer multiply. A program writes the operands to WQ, and writes 1 to AQ to start the process. The multiplier reads the operands from WQ, does the multiply, and writes the 64-bit result to RQ (the low order bits are the first word enqueued). From the write of AQ to a successful (non-stalled) read of RQ takes 10 cycles: two to read WQ, six to do the multiply, and two to write RQ. An implementation using four DSPs could do the multiply in a single cycle, but this doesn’t seem worth doing.

3) A register for miscellaneous output signals is at I/O location 2. Currently only bit 0 of this register is used. It is toggled to generate the “release RS232” signal, which returns control of the RS232 to the TC.

4) The data and instruction caches are in the Dcache module at I/O location 3. The data cache carries out normal operations when AQ [31] is zero. Writing the Dcache as an I/O device allows a range of cache lines to be *invalidated* (the invalid bit for the line is set), or *flushed* (if dirty, the line is written to memory and the dirty bit is cleared). Note that this is an I/O write that doesn’t access WQ. There is currently no way to invalidate the instruction cache, but this could be provided if needed. For these operations, the following AQ fields are used:

   AQ [2:0] = 3, to select the local device
   AQ [9:3] specifies the number of the first line to be invalidated or flushed.
   AQ [16:10] specifies the number of lines to flush – 1. A count of 0 flushes a single line.
   AQ [17] specifies invalidate (1) or flush (0).

5) The “messenger” is at I/O location 4. The messenger provides a way for cores and I/O devices to exchange typed messages of up to 63 words in length. To send a message, a core writes the message payload (n words) to WQ, and writes AQ with the following information:

   AQ [2:0]: 4 (the I/O device)
   AQ [6:3]: The message destination
   AQ [12:7]: The message length
   AQ [16:13]: The message type (the hardware doesn’t interpret the type)

The message is sent on the ring interconnect to the *Message Queue (MQ)* in the destination core. A message on the ring consists of a header with the following fields:

Bits [13:10]: The source core number,
Bits [9:6]: The message type,
Bits [5:0]: The message length in words.

The header is followed by *length* message words.

The receiving core can poll I/O location 4 for messages. If MQ is empty, this read returns zero. Otherwise it returns the message header, with the length in bits 5:0, the type in bits 9:6, and the source core in bits 13:10. The receiving core must then read the n payload words from RQ. MQ is implemented with a 1024-word FIFO, but there is no flow control provided. Software must do this using a higher-level protocol. The details of message passing are described in the discussion of the ring interconnect.

By convention, messages with a length of zero are treated as “control messages” that encode their function in the message Type field. They are not placed in the incoming message queue, but are instead made available to other parts of the CPU (currently only the debug unit) using the output signals ctrlValid(1 bit), ctrlType(4 bits), and ctrlSrc(4 bits). ctrlValid indicates that the other bits are valid; it is asserted for one cycle when a zero-length message addressed to the core arrives on the ring.

6) The lock unit is at I/O location 5. Unlike most systems that implement atomic operations as part of the memory coherence mechanism, this system uses a separate mechanism. There are a system-wide total of 64 mutexes or semaphores, each represented by a bit in a RAM that is part of each lock unit. A particular lock bit may be set in at most one core at any time, but each core may have several different bits set simultaneously.

   A given bit in the lock RAM may be used to provide binary semaphore or mutex semantics, depending on the way the return value of 2 is interpreted.

   To acquire a mutex or do a P (i.e., a Wait) operation, a core reads AQ [2:0] = 5 with the lock number in AQ [8:3]. If the indicated bit is set in the lock RAM, the read returns RQ = 2 immediately. Otherwise, a Preq slot is sent on the ring. If another core has the lock bit set, it converts the lock request into a “Pfail” slot type before forwarding it. If the requesting core receives the unmodified request, it sets the lock bit and returns 1 in RQ. If it receives Pfail, it returns 0 and the lock bit is not modified. To release a mutex or do a V (i.e., a Signal), the core writes AQ[2:0] = 5 with the lock number in AQ[8:3]. The indicated lock bit is cleared. If the lock bit was cleared initially, this is a V operation, and a Vreq slot is sent on the ring. Any core with

the lock bit set clears it upon observing a Vreq from another core. If the bit was set initially, it is cleared and no activity occurs on the ring.

Ring Interconnect

The cores and the memory controller are connected by a ring interconnect. Using a ring reduces routing congestion substantially, allowing the chip to support more cores. The ring is shown in Figure 4:

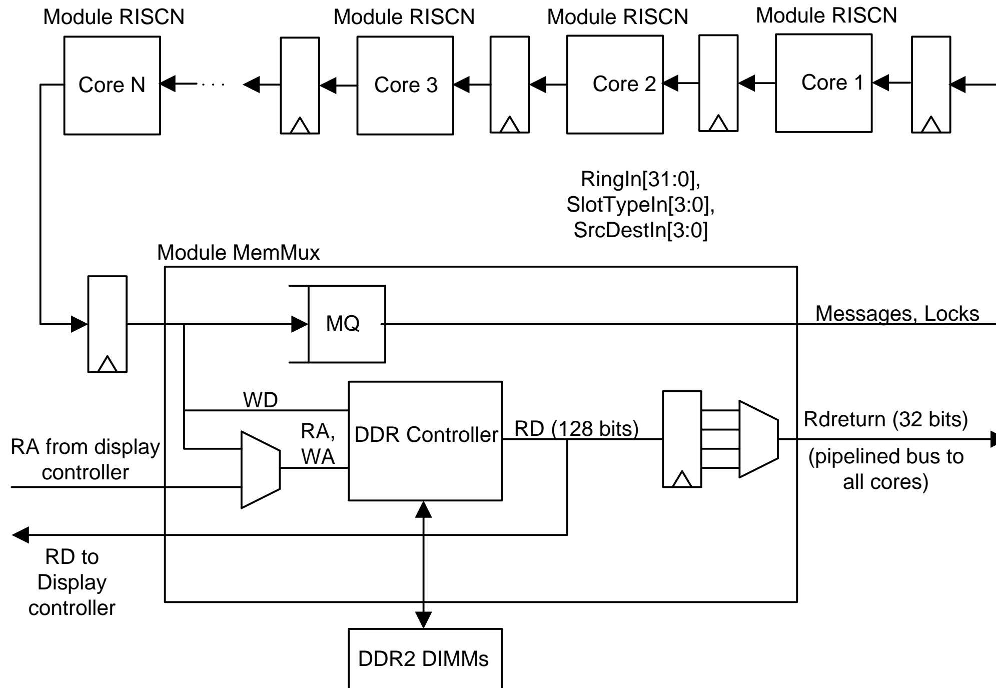

Figure 4: The ring interconnect

The ring consists of three groups of signals: A 32-bit data word, a 4-bit *SlotType*, and a 4-bit *SrcDest* field. There is a register at each node of the ring to pipeline the signals. The logic within each core that interacts with the ring is purely combinational. The DDR controller has been described elsewhere, so its details are not covered here.

The slot types are: *Null, Token, Address, Message, Preq, Vreq, Pfail, and WriteData*.

Read data is returned from the memory on a pipelined bus, the RDreturn bus. This bus carries 32 data bits and a single 4-bit destination field. The bus is pipelined to improve wireability. Message and lock traffic are sent on the primary ring. Using the RDreturn bus improves performance, since it is not necessary to wait for memory data to be sent on the primary ring before emitting another token. There is a "back door" path into the DDR controller from the Display controller (see below). This path inserts read requests into the controller opportunistically, whenever there is a D.C. request and there isn't an address on the ring. A queue (not shown) remembers the source of a read and returns the data either on the RDreturn bus or to the memory controller. The data is sent to the memory controller as 128 bit items.

The memory controller is the master node. When it is able, it emits a single *Token* slot into the ring, with a data value of 0. As the Token passes each node, the node can choose to pass it on immediately if it has no traffic, or to add some number of items to the set of slots that follow the token. In this case, the node passes the data value accompanying the Token plus the number of slots that it needs to send. It then waits until the original number of slots has passed, and adds its data to the ring.

The ring can be likened to a train[8] that starts at a main station and passes through a number of local stations before arriving back at the main station. At each local station, the "station master" (the node) indicates to the driver of the locomotive (the Token) the number of "cars" (slots) it will add to the end of the train, and couples those cars to the train when the last car of the train passes. When the train reaches the main station, the data value accompanying the Token indicates the number of slots that need to be absorbed before the main station can start another train. If the main station didn't wait this number of slots, a second train could collide with cars (slots) being added to the end of an earlier train.

A node adds slots to the train when it services a cache miss, when it sends a *Preq, Vreq* or *Release*, or when it sends a *Message*. Memory data is always transferred in eight consecutive slots, since the memory controller deals in cache lines.

To service a clean read miss a node sends a single *Address* slot. The data consists of a 28-bit line address, and a R/W bit. The *SrcDest* field contains the node's number. The node then waits for *ReadData* slots directed to it, and loads the data (8 consecutive words) into its cache.

When the memory controller sees an *Address* slot, it puts the data into the DDR controller's address queue and if it is a read address, puts the *SrcDest* field into the destination FIFO. When the read completes (the DDR controller does all operations in the order it receives them), the destination FIFO indicates the node that is the destination for the data.

To service a dirty read miss, a node uses ten slots. The first is the read *Address*, the next eight slots are *WriteData* slots containing the dirty line's data, and the final slot is another *Address* slot with the write address.

An eviction of a dirty cache line uses nine slots: Eight *WriteData* slots containing the line's data followed by an *Address* slot containing the write address.

The memory controller puts *WriteData* into its write queue, and *Address* data into its address queue. The controller starts processing a request when it arrives in the address queue.

[8] This is an idealized train. It takes one time unit to pass between stations, and can accelerate instantly when new cars are added.

The message mechanism works as follows: When a node sends a message, it adds the destination/length to the train, followed by the message payload. When a message arrives at the destination node, the node places the data into its Message Queue and replaces the slot type accompanying the message with *Null*. If the message is not absorbed by a slot downstream of the source (i.e., between the source and the memory controller), the memory controller will place the message data and the destination in its MQ, and send it into the ring when the entire train has arrived at the controller. When all messages and lock-related slots have been sent, the controller sends another *Token*, and the process repeats.

Read data is sent on the RDreturn bus whenever it becomes available in the controller's read queue. The RDreturn bus improves read latency, since the controller need not wait for an arriving train to end before sending the data.

Debugging

Debugging of Beehive is under control of the "debug unit". The debug unit is instantiated in all cores, but it is not used by the master core (core 1). The master runs a program that controls the other cores through their debug units. It will probably run a teledebugger stub that communicates with a debugger program running on a separate computer.

To do debugging, we need a way to gain control of a core and cause it to dump its state. It must also be able to reload the state and resume execution. We need to be able to set breakpoints at multiple locations, and determine the PC when a breakpoint occurs. The mechanism described here provides these capabilities.

The Debug Unit

The debug unit contains three registers: savedPC and savedLink are each 32 bits in length; *running* is a single bit.

savedPC and savedLink are loaded when a breakpoint occurs, as described below. *Running* is set when the core is not running and a "start" control message is received from the master, and cleared when a breakpoint occurs or stop control message is received while the core is running. Stop messages to a stopped core are ignored.

The debug unit uses Rw[2:0] to indicate an action taken when J7valid is asserted:

Rw[2:0] = 0: Do nothing.
Rw[2:0] =1: Load link with the address of the "save area" for this core. The save area for a core provides a core-specific block of memory to hold the registers, the PC, the link, and any read queue entries that were present when the core was stopped, plus a count of the number of such entries. The save area for core N starts at (byte) address 0x4000 + 512 * N. Not all words in this region are used.

Rw[2:0] = 2: Load link with savedPC.
Rw[2:0] = 3: Load link with savedLink.

Rw[2:0] = 4: Load link with 0 if the read queue is empty, else 1. This is used to provide a branch operation that tests link for zero in the following instruction. It allows us to access the read queue without stalling on an empty queue. Since this test is not useful in normal code, it seemed wasteful to use a "normal" branch condition for it.
Rw[2:0] = 5: Loads the link with 1 if *running* is true, else 0. This is not useful in normal code.
Rw[2:0] = 6: Indicates that the current instruction is a breakpoint instruction. This condition is set or cleared (in the program being debugged) by the master core.
Rw[2:0] = 7: Reserved.

The master controls a core by sending it a control message (length = 0). A Type 2 control message indicates "kill" (stop immediately, emptying AQ and WQ so that as much of the state as possible can be saved); a Type 1 control message indicates "stop" (stop after executing the next instruction which has Op <= 5, is not liw, and which has a successor instruction that is not a jump[9]); a Type 0 control message indicates "start". When the system is reset, a breakpoint instruction is encountered, or a stop or kill message is received, the core dumps its state and sends the master a message indicating it has done so. It then waits until it observes *running* set by a message from the master, and restores the state from memory and transfers control back to the user code. The details are described below.

When the core is running, the debug unit continuously evaluates the condition (j7valid & (Rw[2:0] == 6)) i.e. "this is a breakpoint instruction" When this condition occurs, the core's state is dumped to the save area as described below, and a "stopped" message is sent to core 1.

When a kill message is received, the same events occur as on a breakpoint or stop, but in addition, the debug unit asserts "emptyAWqueues". This causes the address and write queues to be made empty, allowing the state to be saved correctly. The kill message should be sent only if something goes badly awry. It is not possible in general to cleanly restart a core stopped in this way.

The debug unit provides the value to be loaded into link on the interface signal "linkValue", and indicates that the link is to be loaded by asserting "loadLink".

Stoppable instructions

Saving and restoring the read queue is relatively straightforward. Saving the write and address queues is more problematic. We define a "stoppable instruction" as *an instruction at which, if all outstanding memory and IO operations were allowed to complete, the write and address queues would be empty and the condition codes would be uninteresting*.

By allowing breakpoints only on stoppable instructions, we need not save the write and address queues, and we don't need to save the condition codes. It is the responsibility of the debugger program in the master to set breakpoints only on stoppable instructions. If the debugger wants the core to run without stopping, it can clear all breakpoints and restart the core.

[9] It is possible to restart the core after a "stop", but if the core is stuck in a tight loop, there may be no such instruction. In this case, send the core a "kill" message.

Save/Restore code

The save/restore code starts at location 0. It dumps and restores the machine state. The state includes the registers (r1 – r29), the link, the PC, and a variable number of words that were in the read queue at the time the processor was stopped.

When the save code is finished, the save area contains:

Word 0: The count of the read queue entries saved.
Words 1..29: The registers.
Word 30: the savedLink,
Word 31: the savedPC,
Word 32..32 + count: The read queue values.

The code at location zero saves things as indicated, and then builds the table of instructions used to invalidate the instruction cache when the core is started. It then flushes and invalidates its data cache to ensure the values are written to memory and so that if the master changes them it will receive the correct new values. It then sends a message to the master indicating that it has done so. This is an ordinary message with a one-word payload. The payload is currently (a redundant copy of) the savedPC), but we may find some better use for it[10]. The program then spins waiting for a control message from the master to set *running*. When *running* becomes true, the restore code reloads the registers, the link and PC, and the read queue entries, invalidates its instruction cache, and transfers control back to the user code.

The save/restore code is in Appendix C.

This mechanism allows a non-master core to be started at an arbitrary location with an arbitrary set of register and link values. After a breakpoint, if a debugger program running on the master simply wishes to continue execution on the target, can set the PC to the restart address and send a "start" message.

The master core's instruction cache is preloaded from the FPGA bit stream with the TFTP downloader and the RS232 shell. The start/stop is handled as described in "Testing"

For the non-master cores, the only code with which their instruction caches need to be preloaded is the code described above. When the system is reset, the master core will execute its shell, and the non-master cores will save their state and wait for a "start" message from the master.

Testing

The mechanism can be tested without relying on a downloaded program as the test target. The data caches pre-initialize their dirty bits to "clean". The master core's code reads and writes the first 4KB of memory to itself, to dirty all lines in the cache. The slave code does not do this, so that only locations actually written by the slave will be stored.

[10] It might, for example, return the reason for the stop: breakpoint, stop message, or a wild jump to location 0.

The master code contains a copy of the slave's code assembled into the first locations of its data segment. This is followed by a short test program that contains code with built-in breakpoints and an infinite loop to test stops. Before starting a slave, the master flushes and invalidates its data cache, which places the slave code and the test program into memory. Starting the slave at the address of the test program causes it to execute the program.

The master's pre-initialized instruction cache contains a tiny shell that can be used to do limited debugging. This shell can examine and change memory contents, jump to a location in memory, or control a slave core.

When it is started, it prints its own core number and the number of the current *target core*, followed by ">". It then polls the RS232 receiver and the message queue. When it gets a character, it executes commands or accumulates a numeric value. When it receives a message, it prints " X s Y Z (CRLF)". X is the source core, Y is the message length and Z is the first payload word (currently the target's saved PC). A message with a length > 1 is an error, since we should only see "I stopped" messages.

Numeric input may be in decimal or hex ("xNNN"). There is no backspace, but it is always safe to type "enter" after a typo. This resets the shell and prints the herald. Numbers are accumulated into the internal "value" as they are entered. The commands are single letters or characters:

"value/" *Opens* the memory location "value" and prints the contents in hex. Typing a number followed by "enter" changes the location's contents. "value\" is similar, but prints the contents of the location in decimal.

">" and "<" typed after a location is printed or modified opens the next or the previous location.

"g" jumps to location "value". The data cache is flushed and invalidated before the transfer is done.

"z" runs the TFTP client and loads an image from the server. The IP address of the server is a constant in Master.s.

"t" Sets the targetCore register to "value". If nothing is typed, "value" is invalid and targetCore is not modified). targetCore is set to 2 at initialization.

"j" sets target Core's PC (in the save area) to "value", flushes the data cache to make this value visible, and sends a "start" message to the target. If "value" is not valid, the shell increments the stopped PC by 1 in the save area (the instruction at savedPC (the break instruction) was already executed when the target stopped.

"|" is like "/". It sets addr to the target core's save area plus value, and fetches and displays the contents. Typing a new value followed by CR changes the value. ">" and "<" open the next or previous location (in the save area), as now. Remember that r0 is not stored (location 0 of the save area contains the number of rq entries saved).

"s" sends a "stop" message to the target core.

"k" sends a "kill" message to the target core.

The initial shell can be used to perform limited debugging of downloaded code as well, since unless the master transfers control to the downloaded code using "g", it can start the downloaded program on another core to test it.

When a more capable downloader is written, this will all be retired, but we can use it in the interim to make progress.

High Performance I/O

The system may contain several high performance I/O devices that are shared by the processor cores. These devices appear as nodes on the ring, and interact with the rest of the system using message exchange and DMA. Examples of this sort of device include controllers for the Gigabit Ethernet and the PCI express. Currently, only the Ethernet controller and a high-speed block copy/display control unit are implemented.

To simplify the design of devices, each device includes a simplified version of the CPU described above. This processor doesn't need caches, since the program it runs can be contained in a single Block RAM, and any data structures it uses can be contained in a 1K data memory. The I/O control program is written in assembler. This core includes a local I/O bus, but this bus has only the control registers of the I/O device, a DMA controller and a Messenger. The Ethernet controller core has an RS232, but the block copy/display unit doesn't. Doing I/O controllers in this way is much simpler than building logic to receive, parse, and send messages and operate the device, since much of this complex logic is replaced with software.

Block Copier/Display controller

The Beehive has a copy unit for doing high-speed memory-to-memory copies, and also to provide an interface to the Beehive's display controller. The copier is node EtherCore + 1 on the ring interface. It is shared by the client cores.

The copier does word-aligned transfers, and does not handle overlapping source and destination regions. The copier is started by sending it a three word message:

Word 0: The word address of the source of the transfer (31 bits).
Word 1: The word address of the destination of the transfer (31 bits).
Word 2: The transfer length in words.

When the copier receives the message, it does the copy and returns a 16-bit 1's complement (Internet) checksum of the words transferred to the client in a message.

The message is sent on the ring, but instead of returning control to the client core, the messenger waits for a reply from the copier and loads the checksum directly into the client read queue.

Display controller

The display controller (not present on the BEE3) uses a "back door" into memory to send data to a DVI monitor connected to the XUPv5. The back door is needed because a display can consume more bandwidth than the Beehive's ring interface can support. The display controller uses a hardware DMA engine to transfer pixel data from the DDR2 memory to the display via a Chrontel DVI encoder chip on the board. This chip needs some parameters, which are supplied by the copy unit over an I2C bus immediately after the FPGA is configured.

Programming the display controller is straightforward: A client core sends a message with a one-word payload to the copy unit. The payload is the (cache line) address of a buffer in memory that holds a display frame. The controller acknowledges this message with a single-word reply message with a zero payload. The client stalls until this message is received.

The controller currently only supports monitors with 1280 pixels per scan line, 960 scan lines[11], and 8 bits per RGB pixel. Each pixel occupies a single 32-bit word with bits 31:24 uninterpreted, bits 23:16 = red, bits 15:8 = green, and bits 7:0 = blue.

To avoid "tearing", the controller changes the DMA address only during vertical retrace. The controller guarantees that for each address it receives, at least one full frame will be sent to the display. If the client doesn't send a new buffer address, the last received buffer is sent to the display repeatedly until a new address is received. If the client sends buffer addresses too quickly, the acknowledgement message is withheld until the "one frame per address" guarantee can be met.

Gigabit Ethernet

The Ethernet controller is shown in Figure 5[12]. The controller only supports 1 Gb/sec, full duplex operation. As a result, the sending and receiving sections are almost completely independent. The Xilinx Ethernet MAC is used to handle the details of frame transmission and reception, and the interface to the wire is done by an external PHY chip. The MAC interface provides two 8-bit ports that transfer data at 125 MHz.

---

[11] The monitor can report its native resolution, and the controller can report the size of the connected monitor to the client. This has not yet been implemented.

[12] The ML509 version is slightly different. Since the clock rate of most of the design is 100MHz, but data must be taken from the transmit fifo at 125 MHz, a separate 125 MHz clock (ethTXclock) is provided to read data and header information from the Transmit FIFO. The receive side already handles this transfer properly, since the receive clock is generated by the PHY chip, and is asynchronous with respect to the system clock.

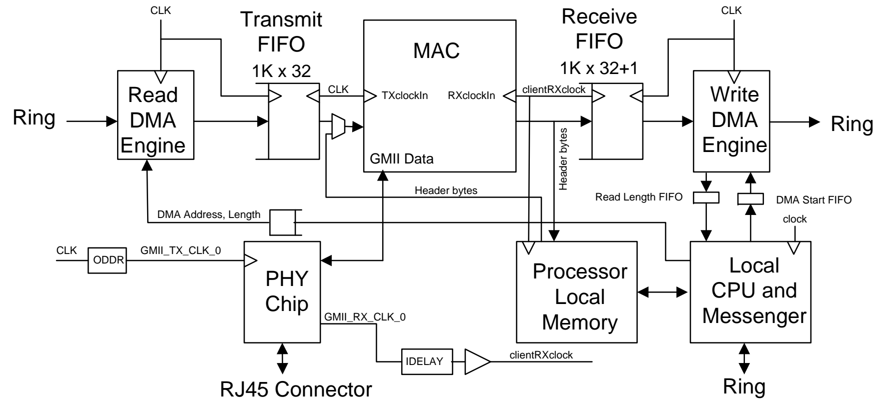

Figure 5: The Ethernet controller. The controller is 1Gb/sec, full duplex only, and employs the simplified clocking scheme of Xilinx UG194 Figure 6-6

The controller contains an I^2C interface that is used to access an EEPROM on the board[13]. The EEPROM is programmed at the time a board is commissioned with the base of a group of 32 MAC addresses. Each user core has a unique MAC address, with the least significant bits equal to the core number. A core acquires its MAC address by exchanging messages with the controller's CPU.

The controller does MAC-address filtering and silently discards frames not directed to one of its client cores. There is limited support for reception of multicast (including broadcast) packets.

Headers sent to the client cores in messages are in network byte order (big-endian), while payload data is taken from memory and transmitted in native (little-endian) byte order. The Ethernet header of a received message is packed into two words of the receive message described below, since the destination MAC address isn't needed. Received payload and header data is placed in memory in native (little-endian) order.

The controller CPU communicates with client cores by exchanging messages with them. There are several message types, differentiated by their length. Message types sent by a client to the controller include:

A) A MAC address request. The length is one word. The controller ignores the message payload, and returns a MAC address reply (see below) to the originating core.

B) An allocation message. The length is two words. The first word is a DMA address (a cache line address) indicating where arriving frames are to be stored in memory, the second is a DMA limit register. A client core indicates its interest in receiving frames by sending this message. The controller will transfer the payload of arriving frames (i.e. the bytes that follow the Ethernet header) into this buffer and increment the DMA address by the number of cache lines needed for the frame. When the

[13] The details are different in the BEE3 and ML509 versions, but the client interface is identical in the two platforms.

limit is exceeded, it is negated, and further frames are silently discarded until a new allocation is received from the client. The client buffer should be one full-sized packet larger than the limit. If the client supplies a negative limit, frames directed to that core are silently discarded.

An allocation message from core 1 is treated differently than allocation messages from other cores. If bit 31 of the DMA limit word is set, the receipt of multicast packets is enabled. Multicast packets are directed only to core 1. When a multicast packet arrives, bit 31 of the fourth word of the receive completion message sent to core 1(see below) to indicate that the packet was a multicast packet.

C) A transmit message. The length of this message is at least four words, and is usually longer, since it includes the payload DMA address, and any headers to prepend to the payload. The first word of the message payload contains the DMA address of the packet payload. This is a cache line address (byte address / 32). Payloads must be cache-line aligned. The second word contains four fields:

Bits [23:19]: The length of the header in words[14].
Bits [18:15]: The number of the core that is initiating the transmission[15].
Bits [14:4]: The length of the payload in bytes.
Bits [3:0]: 0x1[16].

Frames need not end on word or cache line boundaries.

The third word contains the most significant 32 bits of the destination MAC address, and the fourth word of the message contain the low 16 bits of the destination MAC address in bits 31:16, and the EtherType field in bits 15:0.

The remaining words of the message contain any additional headers that are to be sent after the initial Ethernet header. The headers must occupy no more than 27 words, since the received message is copied into a 32-word Transmit Queue entry, one word of which is used for a link. IP and TCP headers without options are 5 words long, but both have a maximum length of 15 words, so we can't hold the maximum sized header, but it is doubtful that these are ever sent.

When a transmit message arrives, the controller copies the message into one of the sixteen Transmit queue entries, providing that one is available and the transmit queue is not full. If not, it rejects the transmit request by sending the initiating core a single-word message with a payload of zero. The core must retry the transmission.

---

[14] The header length does not include the six bytes of the source MAC address, so it will be an integral number of words long (six bytes of destination MAC address, the two-byte EtherType field, and an integral number of additional words for the IP and other protocol headers. These headers occupy an integral number of 32-bit words.

[15] This might not be the core that initiated the message, but usually will be. There may be circumstances in which one core wishes to masquerade as another, and this field provides this option.

[16] This somewhat odd format is used so that the Ethernet controller can load this word directly into its address queue, from which it is loaded into the controller hardware's transmit DMA length FIFO. The "1" is the device address.

If the arriving message is accepted, a single-word message with a nonzero (but otherwise undefined) payload is sent to the initiating core at the time the header is placed in the transmit queue. The frame will be transmitted as soon as its entry reaches the head of this queue. Although the transmit message is sent before the controller has accessed the packet data in memory, higher-level software can ensure there is no race by organizing the packet transmission memory as a ring. If the size of the ring is larger than the size of the transmit queue (4KB), the software knows that the packet will have been sent before it reuses the buffer.

The controller sends three types of messages to client cores, differentiated by length and payload content:

A) A one word message is sent in response to a transmit message. A nonzero payload indicates that the transmit message was accepted for transmission. A payload containing zero is sent if the transmission request was rejected.

B) A two-word MAC address reply message is sent in response to a MAC address request. The first word of the payload contains the most significant 32 bits of the address in the first word and the remaining bits in the low half of the second word.

C) A four-word receive completion message is sent to the client core when a received frame has been placed in memory. The message contains:

Word 1: The frame length in bytes.

Word 2 bits 31:16: The EtherType field. For UDP, these bytes will be 0x0800.

Word 2 bits 15:0: The high order two bytes of the sending MAC address. The high order byte is in bits 15:8.

Word 3: The low order four bytes of the sending MAC address. The most significant byte is in bits 31:16.

Word 4: The cache line address (byte address / 32) of the receive buffer used for the packet.

Packet DMA always writes full cache lines. Any unused bytes after the last byte of a frame are undefined.

Packet transmission can be done without copies, providing that the data to be transmitted is cache-line aligned. Reception involves (at least) one copy, since packet headers and payloads are interspersed in the receive buffer.

The fact that the client caches and memory are not coherent complicates I/O programming. It is necessary for client software to flush any cache lines containing data that will be accessed by the controller before allowing the controller to access the shared memory region. Similarly, a core must invalidate cache lines that will be used by the client to access data placed in memory by the controller.

Control processor specialization

The control processor is similar to the other cores, except for its local I/O system. It doesn't have a multiplier or a lock unit, and its data memory is a 1K word read-only RAM, rather than a cache. It is

word-addressed (unlike the primary cores). Its PC is a 10-bit, rather than a 31-bit value, so the instruction memory is a 1KW ROM. Assembly code for this processor will generated with Basm's –code1, -data1 switches. Assembly code for the other cores uses the -code1, -data4 switches.

The controller CPU also has a local I/O unit (AQ[2:0] = 1) that handles most of the interaction with the controller hardware. This I/O unit has the following components:

1) A receive DMA start FIFO, which has the (28-bit) address into which a received frame should be transferred. This FIFO is loaded from wq by an I/O write with aq[3] = 1.

2) A receive DMA completion FIFO which returns the total number of bytes in a received frame and the good/bad status of the frame, or zero if the completion FIFO is empty. It is read by an I/O read with aq[3] = 0.

3) A transmit DMA address FIFO, which contains the (28-bit) address from which a transmitted frame should be fetched. It is loaded from wq by an I/O write with aq[3] = 0.

4) A transmit DMA length FIFO, which contains the TX DMbase (8),TX header length (5), source core number (4) and the payload length(11) of a frame to be transmited. It is loaded from aq[31:4] by an I/O write with aq[3] = 0.

This unit also has a three bit status field which can be read by an I/O read with aq[3] = 1. The field contains:

1) The RXheaderCountNonzero bit in bit zero. The controller code polls this to determine whether the receiver has deposited the header of an incoming frame in its data memory.

2) The transmit DMA address fifo almostFull flag in bit 1. The transmitter portion of the control program polls this flag to avoid overruning the transmit DMA queues.

3) The receive frameLengthEmpty flag in bit 2. The controller program should only read the frameLength fifo if it is nonempty, or the queue will underflow (*very bad*).

A second local I/O unit, (AQ[2:0] = 5) provides access to two registers, SMACaddressHigh and SMACaddressLow. The concatenation of these two registers yields a 44-bit number that is the most significant bits of the source MAC address. The final four bits are the source core number supplied in tdmaAddrQueue. This register is a constant that depends only on the group of MAC addresses used by the controller. It is initialized at startup by reading the MAC address group from an on-board EEPROM, and not changed thereafter.

Transmission and reception of frames is a cooperative process between the controller hardware and software running in the controller's CPU. First we describe the hardware's role in reception and transmission, then the software's role.

Receive hardware

The receiver hardware is fairly simple. Most of the work is done by the software in the controller CPU.

When a frame arrives from the MAC, it is placed in the Receive FIFO of Figure 5. The first sixteen bytes of the frame, which make up the 14-byte Ethernet MAC header plus two more bytes, are placed in a four word buffer in the processor's data memory. These words contain the destination MAC address (6 bytes), the source MAC address (6 bytes), the Ether Type field (2 bytes), and two undefined bytes. The MAC addresses are placed in DM in big-endian order. When the last word is placed in the DM, a counter that indicates the number of headers that have been buffered in DM but not processed by the CPU is incremented. The CPU polls the nonzero state of this counter (Status register bit 0). When it sees it true, it reads the header in DM, decides where to put the frame in DDR memory, and loads the DMA start FIFO of Figure 5. This register contains the DMA address for the frame. A DMA address of 0 indicates that the frame is to be discarded. This happens if the destination MAC address reported by the hardware is not among the 16 MAC addresses used by the cores on the FPGA. Loading this register also decrements the header counter by 1.

It takes only a few instructions for the CPU to decide where to put the frame. Until the hardware is notified, the packet is buffered in the Receive FIFO. As soon as the CPU supplies the address, the frame is transferred to memory in little-endian order. The DMA machinery can store the packet into memory faster than it can arrive on the wire, so the FIFO will never overflow.

When the frame ends, the Receive FIFO is loaded with an end of frame bit and the frame length (in bytes) plus the GoodFrame and BadFrame status bits from the MAC. When this word arrives at the FIFO output it is placed in the Receive length queue for delivery to the CPU.

Transmit hardware

Transmission is much simpler than reception. In preparation for a transmission, the control RISC puts the header to be transmitted into the CPU data memory, which can be accessed by the DMA machinery.

The control RISC has two local I/O units, one at location 1 (etherQueues.v) and one at location 5 (part of SimpleRISC.v). Other cores do not have these units. The tdmaAddrQ is loaded from WQ [27:0] and the tdmaLengthQ is loaded from AQ[30:4] when an IO write is done with AQ[3:0] = 1. A nonempty AQ starts the transmit process.

The 28-bit tdmaLengthQ entry is composed of four fields:

1) Bits 10:0 are the payload length in bytes.

2) Bits 11:14 is the core number of the core that initiated the transmission. This is used as the last nibble of the source MAC address when the packet is transmitted. It is also used to control which core receives a “transmit completed” message from the controller.
3) Bits 19:15 are the header length in words, not including the source MAC address, which is constructed by the controller. For a naked Ethernet packet, this length should be 2 (six bytes for the destination MAC address, two for the EtherType field).
4) Bits 26:20 are the controller data memory location from which the packet header will be sent. The base read address is 8 * tdmaLengthQ[26:20] + 3. This counter is incremented as header words are read from the data memory.

When transmission starts (tdmaAddrQ nonempty), the tdmaAddrQ entry is placed into the DMA address FIFO, and the tdmaLengthQ entry is placed in the Transmit FIFO of Figure 5. The transmit DMA machine reads the number of words needed for the packet from the CPU’s data memory into the Transmit FIFO. At the FIFO output, the packet length is compared with the number of words in the FIFO. When the entire packet has been read, the packet is transmitted as an uninterrupted stream of bytes. First, the packet header is extracted from the control CPU’s DM. This is six bytes of destination MAC address, followed by six bytes of source MAC address, followed by the rest of the header. As soon as the requisite number of header bytes has been sent, the machine sends the balance of the frame from the Transmit FIFO, and idles awaiting another transmission.

Ethernet Controller Software

The Ethernet controller RISC uses its 1K data memory to hold three structures:

1) An array of two-word entries containing a DMA address and a DMA limit for each core. This array starts at location 128d, and is indexed with two times the low order 4 bits of the Source MAC address of an arriving frame (these bits indicate the core that is to receive the frame).
2) A group of 16 32-word Transmit queue entries. These are managed as a ring, as shown in figure 7 below. These buffers occupy locations 256d-767d.
3) A set of 64 4-word receive header buffers. These buffers are used cooperatively as a ring by the receive hardware and the receive process in the controller software. The ring can never overflow, since the hardware maintains a count of the number of occupied buffers, and if this counter exceeds 31, arriving frames are silently discarded. For this situation to occur, the controller would need to see a stream of minimum-length frames, which it shouldn’t see. The pointer into the buffer is initialized to 768d. When it reaches 1024d, it is reset to 768d.

The software for the controller is a loop that handles both transmission and reception of frames. There are no interrupts involved. The flow is shown in figure 6:

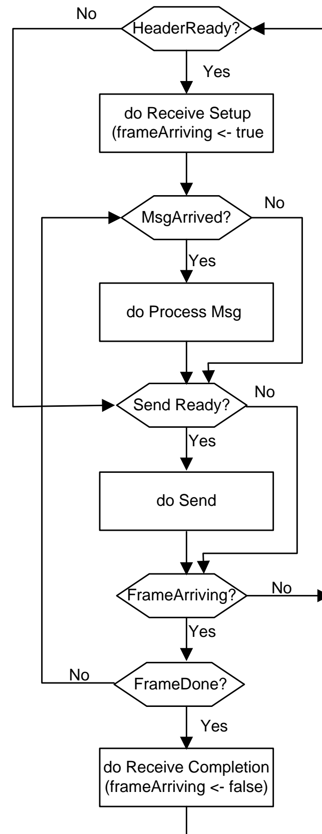

Figure 6: Ethernet controller software flow.

There are five primary tests and four activities, each of which can be done in a few instructions.

When a header arrives (Status.headerCountNonZero), the Receive Setup activity first tests that the packet is for a destination served by this controller (i.e. the 44 most significant bits of the MAC address in the received header is in the controller's MAC address range). If it is not, the packet is discarded. It then loads the DMA address into the DMAstart FIFO (Figure 5). The DMA address comes from the read address array in DM indexed by the low 4 bits of the received MAC address. This table contains a cache line address for the data and a limit addresses. The frame is accepted only if the address is non-negative, otherwise, it is discarded. Frames are discarded by giving the hardware an address of zero. When Receive Setup has sent the DMA address, it sets frameArriving to indicate that a frame is arriving.

The Receive Completion activity runs when the readLength FIFO is loaded by the hardware. The FIFO has the frame payload length in bytes, and the header length is in the headerLength FIFO (this must be a FIFO, since several frames may arrive while the Receive FIFO is being emptied). The Receive Completion task does the housekeeping needed to complete the arrival process, including updating the DMA address, and sending a message to the core which is the destination of the frame. If the new DMA address exceeds the DMA address limit, the DMA address is negated to cause further frames to be dropped. Request-response protocols that expect a single frame (e.g. DHCP) can set the limit to a value smaller than the minimum frame (but must still provide a buffer large enough to hold the largest

possible frame). Finally, the activity clears frameArriving, which allows the "Frame Arrived?" test to be executed on the next loop iteration.

The Process Msg activity polls the controller's message queue for incoming messages. There are three types of message: A single-word message that requests a MAC address (the message payload is ignored), a Read message which contains DMA start and limit addresses (which are cache line addresses, i.e. word addresses / 8), and a Write message that contains a DMA address for transmitted frames for the source core, the message length in bytes, the header length in words, the destination MAC address and the EtherType field, and up to 27 words for other headers. The message payload is copied into the Transmit queue of Figure 7, starting at the word after the link word, provided that doing so would not cause the queue to overflow. If overflow would occur, the controller returns a "writeReject" message is returned to the sender, which must try again later. When the message payload has been copied into DM, TxTail is advanced.

The payload of a Read message is copied into the array entry in the read address array mentioned above. Although the operations of each activity are atomic with respect to other activities, there is a complication, since if a Receive message arrives carrying a new buffer allocation while a packet is arriving for the new buffer's core, then the address should *not* be updated with the packet length when the arrival completes. The Process Msg task sets a flag bit in a register when a receive message (new buffer allocation) arrives for a core that is the destination for an arriving frame. The Receive Completion task tests the flag (and clears it if it is set), and updates the DMA address only if the flag was clear. If the receipt of a frame causes the address to exceed the limit, the address is set to -1, which causes subsequent frames to be discarded.

The frame is not transmitted at the time the Write message arrives. Instead, the contents of the message are copied into a circular list of pending transmissions. If this queue is full at the time a Write message arrives, a "writeReject" message is returned to the requesting core.

The structure of the Transmit queue is shown in Figure 7. Each entry corresponds to a Write message. TxHead and TxTail are held in CPU registers. TxHead points to the next message that will be sent, TxTail points to the next available (empty) Transmit queue entry. Since the individual entries are large, there may be fewer entries than cores (which is why Write messages can be rejected).

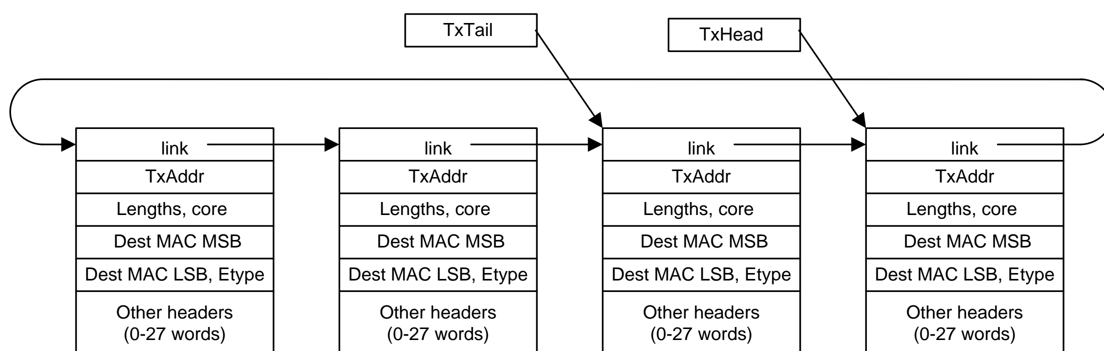

Figure 7. The transmit queue. The actual queue contains 16 entries.

The Send action is invoked when the transmitter's command/address FIFO is non-full and the Tx queue is nonempty (TxHead != TxTail). The Send process loads the command/address FIFO with the TxAddr, Payload Length, and Header Length, plus a pointer to the Dest MAC MSB field of the command queue (TxHead – the DM address is incremented by 3 by the controller hardware to point to the Dest MAC MSB when the transmission takes place).

System Software

Currently, a simple assembler and a C compiler based on GCC are available, thanks to Tom Rodeheffer. Tom is also building an MSIL compiler that can be used to compile C# programs.

Bootstrap programs are assembled with Basm (the assembler) and patched into the bitstream files used to configure the FPGA. The script "MakeX.cmd" in the "scripts" directory is used to assemble the necessary files and patch them into the FPGA bitstream (X is the number of cores). This limits initial programs to 1024 words of data and 1024 words of instructions, which suffices for testing and initialization.

Two files are used for the non-specialized cores: Master.s is used for core1, and Slave.s is used for the remaining cores. Master.s includes an interactive shell described in the section on debugging, and a simple DHCP client, a routine to send and receive ARP packets, and a TFTP client to download programs from a TFTP server. The IP address of the server is a constant in this program. The filename requested from the TFTP server is a 4-character string generated by adding "a" to each of the four nibbles that comprise the low bits of the sending core's MAC address. This allows each core to get a unique boot file. The downloaded program is placed at location 0x4000. Control is transferred to it by typing "x1000g" to the shell. The network routines do essentially no error checking, but this has not proved to be a problem. Slave.s includes only the debugging code to stop and start the core.

The TFTP server cannot be run on a domain-joined machine, since such machines exchange packets using IPSec, which the FPGA program can't handle. We use a dedicated PC for this purpose. It is possible to use RDP to connect to this machine, so it can receive files from a domain-joined machine. A packet sniffer (e.g. Wireshark) is invaluable when debugging interactions between the board and the TFTP server.

The program EtherCore.s is the code for the Ethernet controller's control RISC (core nCores). This code includes a shell and the code necessary to run the Ethernet controller. The program reads the base MAC address for the FPGA's cores from an EEPROM as part of initialization. By default, the shell is bypassed when the code begins executing after FPGA configuration, and the program jumps directly to the controller code. This can be modified if this program needs the shell for debugging, or when the shell is used during commissioning of a new board to set the MAC address base. The actual MAC address used is the base address plus the core number. To enable the shell, patch out the "j   resetPhy" instruction at shell+5 in EtherCore.s, and run the patching script. The correspondence between location names and addresses can be determined by inspecting the listing file EtherCore.lst produced by BAsm. When the

bitstream is loaded into the FPGA, the shell will be present when you select the Ethernet core with the TC shell (core 4 or core 14, depending on the number of cores generated).

At Microsoft, Beehive MAC addresses are assigned in groups of 32 addresses for each FPGA (XUPV5 or BEE3), allowing each FPGA to have 32 MAC addresses.

A shell routine is provided to write an address supplied by the user into the EEPROM. Using the console, store the MAC address in locations MacBase and MACBase + 1 and execute the code at “initMAC”. MacBase is location 0x12, initMac is ~0x176[17]. The low order four bits of the MAC address must be right-shifted by four, so if the (hexadecimal) Mac base address is ab-cd-ef-gh-ij-k0, type:

```
4>x12/ 0 xabcdefgh <enter>
4>x13/ 0 x00000ijk <enter>
4>x176g //the “g” is not echoed.
4>
```

This will write the MAC address into the first 8 bytes of the EEPROM. The write can be verified by printing out the EEPROM contents a byte at a time. Type “np” (n is the byte address) to see each byte.

In the XUPv5 version of the system, the bitstream supplied to the FPGA can be taken from a Compact Flash card on the board. The CF card can be loaded during board commissioning using the Xilinx Impact tool. This allows the user of the board to use the system without having installed any of the Xilinx tools on their PC. We use a CF card with a single .ace file on it, rather than the more complicated file system suggested by Xilinx (which appears not to work).

In addition to setting the MAC address, commissioning a new XUPv5 board involves several things:
1) The 256 MB SODIMM supplied by Digilent must be replaced with a 2GG SODIMM. These are available from several vendors. We use 667 MHz parts for greater margins.

2) It is a good idea to put a heatsink and fan on the FPGA. In normal operation, the chips get quite hot without this. Digilent suggests a proper fan in their literature.

3) Optionally, put the FPGA bitstream into the XUPv5’s Compact Flash card.

Conclusions

Unlike many other architecture projects, it is not our aspiration to run complete operating systems. We plan to use the machine to experiment with hardware features that improve our ability to program many-core systems. For this, a simple system that can support many cores at reasonable performance should be sufficient.

[17] This address may change in future versions, so it is a good idea to look it up in the listing file (EtherCore.lst in the basm directory).

Appendix A: The TC shell

TestTC is a simple shell for TC RISC in the memory controller. It also contains routines to initialize and calibrate the DIMMs, and refresh the memory. It communicates with the (human) user over a 115,200 baud serial link of 8 data bits, 1 stop bit, no parity, and no flow control. The BEE3 control board brings out the serial link as a USB port, so the host must have the appropriate driver. The XUPv5 board provides a standard 9-pin connector which can be connected to a PC using a null modem cable; HyperTerminal is a useful utility to run the shell.

The TC is quite small. It has a 1K instruction ROM which is initialized with a so-called .coe file, rather than using the patching script the other cores use. Changing the .coe file requires that the core be regenerated and the design rebuilt, but this shouldn't happen frequently. The TC has a 64-word data memory, currently used only for the stack.

When the board is reset or the FPGA is reconfigured, the TC starts execution at location 1. It resets the system, waits for 200 us as required by the RAMs, initializes the RAMs on the four DIMMs..

It then attempts to calibrate the DDR2 DIMMs, and starts a refresh timer. If this succeeds, it prints:

```
s
>
```

If calibration fails, it prints:

```
f
>
```

This is dire, and requires debugging, probably with ChipScope. It may mean that the DIMM is not properly inserted in its socket.

Once the shell is started, it listens for input. Digits are assembled into the "current value" as they are typed. All input is hexadecimal (0..9, a..f). Other characters recognized by the shell are:

"/": This "opens" the location in the register file given by the current value, and prints the contents. Typing "/" again uses the contents as an address, opens that cell, and prints the contents (not very useful).

">": Opens the next location in the data memory and prints its contents.

"<": Opens the previous location in the data memory and prints its contents.

"Enter": This "closes" the currently open cell by storing the current value into it. To read the contents of location 256 (say 3), change it to 20, and store it, the output is:

```
>256/00000000003 20 (enter)
>
```

"g": This jumps to the ROM location given by the current value, saving the return link on the stack.

Typing:

```
>1g
```

Resets and reinitializes the system, and restarts the shell.

"ns": (n a number from 1 to nCores): This switches control of the RS232 line to core n . The "s" is not echoed.

The shell simply skips over unrecognized characters without echoing them. There is no backspace facility, but typing "/" "enter" if an input value is mistyped is usually safe. If you jump to a location above the end of occupied instruction memory, the shell will restart, since the TC will happily increment the PC and roll over to the initialization sequence. If you jump into a random location in occupied IM, anything can happen.

Appendix B: Design Module Hierarchy

The table below shows the module hierarchy for the design, with the approximate size in lines of the non-trivial modules. The entire design is approximately 5000 lines of Verilog.

```
RISCtop.v (RISCtop) [300]
  |--RISC.v (riscN) [450]
  |     |--DebugUnit.v (debugger) [80]
  |     |--rs232.v (rs232x) [100]
  |     |--mul.v (mulUnit) [150]
  |     |--newestDCache.v (dCacheN) [275]
  |     |     |--instCache.v (instCache)
  |     |     |--tagmemX.xco (dataTag)
  |     |     |--dValidTag.xco (dataInvalid)
  |     |     |--itagmemX (instTag)
  |     |     |--dpbram32.v (dataCache)
  |     |--Messenger.v(msgrN) [200]
  |     |--Locker.v (lockUnit) [125]
  |     |     |--lockMem.xco (Locker)
  |     |--regFileX.xco (RFa)
  |     |--regFileX.xco (RFb)
  |     |--XmaskROM.xco (masker)
  |     |--PipedAddressQueue.v (addressQueue)[100]
  |     |     |--dpram64.v (qram)
  |     |--queueN.v (writeQueue)
  |     |     |--dpram64.v (qram)
  |     |--PipedQueue32nf.v (readQueue)
  |           |--dpram64.v (qram)
  |--memMux.v (mctrl) [300]
  |     |--queueN.v (destQueue)
  |     |     |--dpram64.v (qram)
  |     |--TinyComp.v (TC5, timex) [375]
  |     |     |--dpbram36.v (im)
  |     |     |--dpbram36.v (rfA)
  |     |     |--dpbram36.v (rfB)
  |     |     |--rs232a.v (user)
  |     |--ddrController.v (ddra) [600]
  |           |--AF.v (addrFifo) [50]
  |           |--dpram.v (rInhibit)
  |           |--camx.v (OB) [100]
  |           |     |--dpram.v (rowElement)
  |           |--WB.v (writeBuf)
  |           |--RB.v (readBuf)
  |           |--ddrBank.v (ddrBankx) [200]
  |                 |--dqs_iob.v (dqsPad0) [50]
  |                 |--dq_iob.v (dqx) [150]
  |--Ethernet.v (ethcon) [600]
  |     |--queueN.v (rsStageFifo)
  |     |     |--dpram64.v (qram)
  |     |--SimpleRisc.v (controlRisc) [450]
  |     |     |--rs232_I2C.v (rs232) [100]
  |     |     |--etherQueues.v (ethQ) [150]
  |     |     |     |--queueN.v (rdmaAddrQ)
  |     |     |     |    |--dpram64.v (qram)
  |     |     |     |--queueN.v (tdmaAddrQ)
  |     |     |     |    |--dpram64.v (qram)
```

```
   |    |    |    |--queueN.v (tdmaLengthQ)
   |    |    |    |   |--dpram64.v (qram)
   |    |    |    |--queueN.v (rxFrameLengthQ)
   |    |    |        |--dpram64.v (qram)
   |    |    |--SimpleDataMemory.v (dCacheN) [100]
   |    |    |    |--dpbram32.v (dataCache1)
   |    |    |    |--dpbram32.v (dataCache2)
   |    |    |--Messenger.v (msgrN)
   |    |    |--rom32.v (instMem)
   |    |    |--regFileX.xco (RFa)
   |    |    |--regFileX.xco (RFb)
   |    |    |--XmaskROM.xco (masker)
   |    |    |--queueN.v (adddressQueue)
   |    |    |    |--dpram64.v (qram)
   |    |    |--queueN.v (writeQueue)
   |    |    |    |--dpram64.v (qram)
   |    |    |--queueN.v (readQueue)
   |    |         |--dpram64.v (qram)
   |    |--MAC.v (etherMAC) [250]
   |    |--EthWriter.v (ewriter) [150]
   |    |    |--EthfIFO.xco (writeFIFO)
   |    |--EthReader.v (ereader) [200]
   |         |-transmitFifo.xco (txFifo)
   |--RISC.cdc
   |--DDRA.ucf
   |--Ethernet.ucf
```

Appendix C: Debugging Save/Restore code

```
.code
/* The slave stop/start code.

When the system is reset, the program starts at location 0,
waits 10 seconds, then dumps the state. The copy of the code
that the master puts in memory when restarting the slave
doesn't have this delay.  It is provided to give the user
time to type "1s" to give the RS232 to the master before
the state is dumped.

The code uses only self-relative branches, since it will
also be assembled into the data segment of the master, and
named branch destinations would have the wrong offset.
*/

void     = $0
zero     = $0
.assume  zero, 0
count    = $1
base     = $2
linkAddr = $3
addr     = $4
rqCount  = $5

invalWordAddress = 0x1ffffc00 //(2GB - 4KB)/4.  used for call
invalByteAddress = 0x7ffff000 //used when building the table

// inv=0 cnt=127 line=0 dev=3
FLUSHALL = (0 LSL 19) + (127 LSL 12) + (0 LSL 5) + (3 LSL 2) + 2

// inv=1 cnt=127 line=0 dev=3
INVALALL = (1 LSL 19) + (127 LSL 12) + (0 LSL 5) + (3 LSL 2) + 2

// type=1 len=1 dst=1 dev=4
```

```
STOPACKMSG = (1 LSL 15) + (1 LSL 9) + (1 LSL 5) + (4 LSL 2) + 2


 ld       zero, 0 //nop at location 0.

 //long_ld  link, 300000000  //wait 10 seconds
memLoop:
  //sub      link, link, 1
  //jnz      .-1

  ld       wq, $1  //put the registers in WQ
  ld       wq, $2
  ld       wq, $3
  ld       wq, $4
  ld       wq, $5
  ld       wq, $6
  ld       wq, $7
  ld       wq, $8
  ld       wq, $9
  ld       wq, $10
  ld       wq, $11
  ld       wq, $12
  ld       wq, $13
  ld       wq, $14
  ld       wq, $15
  ld       wq, $16
  ld       wq, $17
  ld       wq, $18
  ld       wq, $19
  ld       wq, $20
  ld       wq, $21
  ld       wq, $22
  ld       wq, $23
  ld       wq, $24
  ld       wq, $25
  ld       wq, $26
  ld       wq, $27
  ld       wq, $28
  add      wq, zero, $29  //r29 (via a port) is the read queue.  Use the b port

//put savedLink and savedPC into wq
  j7       3        //link <- savedLink
  ld       wq, link
  j7       2        //link <- savedPC
  ld       wq, link

//issue 31 writes to store r1-r29, savedLink, and savedPC to words 1..31 of the save area.
  ld       count, 31
  j7       1        //link <- save area
saveRegs:
  aqw_add  link, link, 4
  sub      count, count, 1
  jnz      .-2

  ld       linkAddr, link  //linkAddr is used for subsequent addressing

//copy rq entries into wq and write them to memory until rq is empty
//count the number of entries with count (now = 0)
rqToWq:
  j7       4    //link <- 0 if rq empty, else 1
  ld       void, link  //test it
  jz       .+5
```

```
  ld       wq, rq
  aqw_add  linkAddr, linkAddr, 4
  add      count, count, 1
  j        .-6

rqSaved:
  ld       wq, count  //save count in word 0 of the save area
  j7       1   //link <- save area
  aqw_ld   link, link

//save is now complete

//build the table of instructions at invalIcache
long_ld    addr, invalByteAddress
sub        addr, addr, 32
ld         count, 127  //first 127 entries (8 words apart) get the jump instruction

buildTable:
long_ld    wq, 0xf800220c //j .+8
aqw_add    addr, addr, 32
sub        count, count, 1
jnz        .-3
long_ld    wq, 0xf000030c  //j link
aqw_add    addr, addr, 32

// flush and invalidate the data cache

  aqw_long_ld  void,FLUSHALL
  aqw_long_ld  void,INVALALL

// send a message to the master, indicating we've stopped
       j7      2  // link <- savedPC
       ld      wq,link      // message payload
       aqw_long_ld  void,STOPACKMSG

// -------------------------------------------------------------
// wait for the master to start us
// -------------------------------------------------------------

waitStart:
  j7     5  //link <- 1 if running, else 0
  ld     void, link   //test it
  jnz    .+2
  j      .-3

restore:
  j7       1 //link <- save area
  sub      link, link, 4
  ld       count, 32  //issue reads for count, registers 1-29, savedLink, savedPC
readRegs:
  aqr_add  link, link, 4
  sub      count, count, 1
  jnz      .-2

reloadRegs:
  ld       $28, rq    //put RQ count into r28
  ld       $1, rq     //load registers 1-27, r29
  ld       $2, rq
  ld       $3, rq
  ld       $4, rq
  ld       $5, rq
  ld       $6, rq
```

```
  ld       $7, rq
  ld       $8, rq
  ld       $9, rq
  ld       $10, rq
  ld       $11, rq
  ld       $12, rq
  ld       $13, rq
  ld       $14, rq
  ld       $15, rq
  ld       $16, rq
  ld       $17, rq
  ld       $18, rq
  ld       $19, rq
  ld       $20, rq
  ld       $21, rq
  ld       $21, rq
  ld       $23, rq
  ld       $24, rq
  ld       $25, rq
  ld       $26, rq
  ld       $27, rq


//rq now has r28, r29, savedLink, savedPC
//$28 contains the number of RQ entries to restore.
//We use it because r29 isn't a good place to stand
//continue reading until $28 < 0.
loadRQ:
  sub      $28, $28, 1 //all RQ entries read?
  jm       .+3
  aqr_add  link, link, 4
  j        .-3

rqDone:
//rq now contains r28, r29, savedLink, savedPC, and the RQ entries.

//invalidate the instruction cache
 long_call invalWordAddress

 ld        $28, rq    //load r28
 ld        $29, rq    //load r29
 ld        link, rq   //load link
 j         rq         //transfer to user code.
```