

# Concurrency in Multi-Core Processor Design

BY PATRICK CLANCY

Haverford College  
advised by David G. Wonnacott

## Abstract

It is possible to extend a microprocessor from a single core to a multiple cores by replicating the single core processor, and interfacing them to main memory bus via a bus arbitrator. With a multi-core processor, the possibilities for parallel programs are apparent, but the programmer must overcome those obstacles with lock based programming. Transactional Memory would provide a means for programmers to handle highly concurrent programming in a more forgiving environment. This thesis surveys these topics and discusses possible implementations in terms of the HERA architecture.

## Table of contents

<b>Table of contents</b> . . . . .	2
<b>List of figures</b> . . . . .	3
<b>1 Introduction</b> . . . . .	3
1.1 HERA Philosophy . . . . .	3
1.2 Multi-core vs. Multi-CPU . . . . .	3
1.3 Bus Arbitration . . . . .	4
1.4 Caches . . . . .	4
1.5 Transactional Memory (TM) . . . . .	4
1.5.1 Software and Hybrid Transactional Memory Systems . . . . .	4
<b>2 Multi-core Processor Design</b> . . . . .	5
2.1 Bus Arbitration . . . . .	6
2.1.1 Organization . . . . .	6
2.1.2 Identifying a Problem . . . . .	7
2.1.3 Selecting Among Multiple Bi-directional Bus Lines . . . . .	8
2.1.4 Scheduling/Fairness . . . . .	8
2.1.5 Correctness . . . . .	9
2.1.6 HERA Multi-core Design and Departures from Optimal Design . . . . .	11
<b>3 Concurrency</b> . . . . .	12
3.1 Example Problem . . . . .	12
3.2 Lock Based Solution . . . . .	12
3.3 General Limits of Concurrency . . . . .	13
<b>4 Extending a HERA multi-core CPU to support Transactional Memory</b> . . . . .	13
4.1 Overview . . . . .	14
4.2 Transactional Cache . . . . .	14
4.2.1 Associative Memory Cells . . . . .	14
4.2.2 Fully Associative Cache . . . . .	15
4.2.3 Interfacing the Transactional Cache with the HERA multi-core CPU . . . . .	15
4.3 Transactional Instructions . . . . .	16
4.3.1 Implementing Transactional Instructions . . . . .	16
<b>5 Dining Philosophers Example</b> . . . . .	16
5.1 Problem Definition . . . . .	16
5.2 Lock Based Solution . . . . .	17
5.2.1 Discussion of Tanenbaum’s Solution . . . . .	19
5.3 Transaction Based Solution . . . . .	19
5.3.1 Discussion of Transactional Solution . . . . .	20
<b>6 Conclusions</b> . . . . .	21
<b>Bibliography</b> . . . . .	22

## List of figures

The HERA multi-core CPU main module. . . . .	5
A HERA quad-core CPU. . . . .	6
Organization of Bus Arbitrator module. . . . .	7
Circuit identifying if two or more CPU cores require access to the bus. . . . .	7
Bus arbitration enable logic. . . . .	8
Orientation of tri-state buffers used to correctly select bi-directional data lines. . . . .	8
Simple counting circuit used to select which CPU core has access to main memory. . . . .	9
Logic added to clock lines so that CPU cores can be “halted”. . . . .	9
Oscilloscope output used as timing diagram. . . . .	10
Segment of a HERA assembly language program corresponding to the diagram in Figure 9. . . . .	10
More specific timing diagram for Figure 8. . . . .	11
Coincidental writes to the same location in memory. . . . .	12
By adding an Exchange instruction, we can solve the problems in Figure 12 [Tan01, Sec 2.3.2]. . . . .	13
An associative memory cell [Hay88]. . . . .	14
A 4x4 bit associative memory cell[Hay88]. . . . .	15
Additions to the HERA architecture. . . . .	16
Four philosophers, four plates of noodles, but only four chopsticks. How can everybody eat? . . . . .	17
A solution to the dining philosophers problem [Tan01]. . . . .	18
Pseudo-code for a transactional solution. . . . .	20

## 1 Introduction

In the world of modern micro-processors, single-threaded performance is staying relatively constant, while most of the effort goes into multi-threaded systems. In order to make performance gains, multi-core and multi-processor systems are becoming, if not already the standard. An interesting question to look at, is how to build a multi-core processor if you already have a single-core processor. Specifically, the question of interest is how can a HERA[Won06] processor be extended to support multiple cores. A multi-core system puts the burden of highly concurrent programs on the programmer. With an increased desire for parallelism, the programmer must handle the inevitable problems with locking. In order to avoid these problems, a lock-free solution must be used, one novel solution was introduced by Herlihy and Moss, transactional memory[HM93]. Transactional memory (TM) involves mostly hardware modifications, including an associative transactional cache and modifying the HERA instructions.

### 1.1 HERA Philosophy

The philosophy of the HERA architecture can be mostly explained with the following statement: make everything as simple to possible to implement, while still performing the operations of interest. In other words, performance is not a concern. The goal is to see how things are built, and do them as easily as possible.

### 1.2 Multi-core vs. Multi-CPU

What’s the difference between a multi-core CPU and a multi-CPU system? Physically, a multi-core CPU and a multi-CPU system differ because the multi-core version has more than one CPU inside a single chip, while the multi-CPU system uses more than one chip. Both methods of multi-processing have that advantage; if properly implemented, they provide an increase in per-

formance over a single-core or single-CPU system. Since a multi-core CPU is only one chip, the connection between CPU cores is physically much smaller, so the information has to travel less distance, and therefore, takes less time to reach its destination. A multi-core CPU uses substantially more complicated circuitry which requires more physical space, but it is contained in a chip that is roughly the same size. A multi-CPU system has the advantage that each CPU will run cooler, and require less energy to remove excess heat than a multi-core CPU. Less heat makes each CPU run more efficiently, but each CPU in the multi-CPU system needs to be individually powered, whereas the multi-core CPU needs substantially less power, because there is only one physical chip in the system. In extending the HERA CPU, we will use the multi-core architecture, meaning that in the main CPU module, there will be more than one HERA CPU core.

### 1.3 Bus Arbitration

In a multi-core CPU, what happens if CPU core #1, CPU core #2, and CPU core #3 all want to access main memory at the same time? Since we're not using multi-port memory, only one CPU core can access main memory at any given time. Only one CPU core may access the bus lines to main memory, otherwise the 16-bit value on the bus will be undefined during a memory operation. The method by which a CPU core is given access to the bus lines is called *bus arbitration*, and it is essential to the extension from a single-core CPU to a multi-core CPU.

### 1.4 Caches

Since the HERA multi-core CPU has one piece of main memory, RAM, read/write operations will significantly reduce the speed at which the multi-core system can operate. The goal of the multi-core CPU is to have each core operating simultaneously, but if our options for reading and writing from memory are limited to just RAM, it's clear that our performance will suffer. An addition to our CPU that would improve performance would be implementing a cache. Interfacing a cache in between the CPU and main memory would decrease the number of times reads/writes from main memory. For the HERA multi-core system, we chose not to implement a cache, because the CPU operates well without the cache, besides the fact that it is slow. This choice is consistent with the design philosophy.

### 1.5 Transactional Memory (TM)

The seminal paper written on Transactional Memory written by Herlihy and Moss[HM93] is still the primary reference. They define lock-free data structures as those which "do not require mutual exclusion." Lock-free can avoid the following problems in a lock-based design: priority inversion, convoying, and deadlock. In addition to avoiding these problems, TM systems typically involve less memory accesses, and have the ability to outperform conventional lock-based systems. To do so, transactions must be both serializable and atomic. Implementing TM requires significant hardware modifications: additional transactional instructions, a private transactional register, an additional fully associative transactional cache, and the transactional cache must snoop on the main bus. In general, the TM system tentatively writes to the transactional cache, and decides whether or not to commit based on whether or not another transaction has touched its word in memory. The HERA architecture can be extended to support TM, and it is the basis of Section 4.

#### 1.5.1 Software and Hybrid Transactional Memory Systems

Shavit and Touitou[ST95] introduce software based transactional memory (STM), using the transactional approach set forth by Herlihy and Moss. STM has the advantage that it requires no hardware modifications, so it is possible to implement on any given hardware, by adding a library

to some given programming language. STM is also capable of dealing with larger data structures, where TM is limited by the size of the transactional cache. It still achieves the same goal: making highly concurrent programs easier to write, subsequently taking advantage of the increased parallelism of a multiprocessor system. In addition to using lock-free data structures, like Herlihy and Moss, Shavit and Touitou’s software transactional memory is non-blocking. Where non-blocking means: “the repeated execution for some transaction by a process implies that some process will terminate successfully after a finite never of attempts in the whole system.” This approach is guaranteed to terminate after a finite number of steps, so it is both non-blocking and wait-free. Damron, Fedorova, and Lev[DFL06] present a new system that dynamically selects between a TM system and a STM system depending on the particular circumstances. This approach is called Hybrid Transactional Memory (HyTM), and it’s goals are identical to the previous TM systems: reducing the difficulty of writing correct concurrent programs. HyTM uses a best effort TM system that does not necessarily have to be implemented, for HyTM to work, because there is a STM system that can be used in all cases. The TM system is present for the cases in which TM can outperform STM, which for the most part will be concurrent programs using smaller data structures. Both STM and HyTM could be implemented with the HERA architecture, and a compiler that produces HERA assembly language, but those modifications are beyond the scope of this paper.

## 2 Multi-core Processor Design

For this project, the decision was made to extend a HERA CPU from a single to core to four cores. The process of converting a single core HERA CPU to a multi-core CPU involves making multiple copies of each CPU, and then arbitrating the bus lines that travel between each CPU core and main memory.

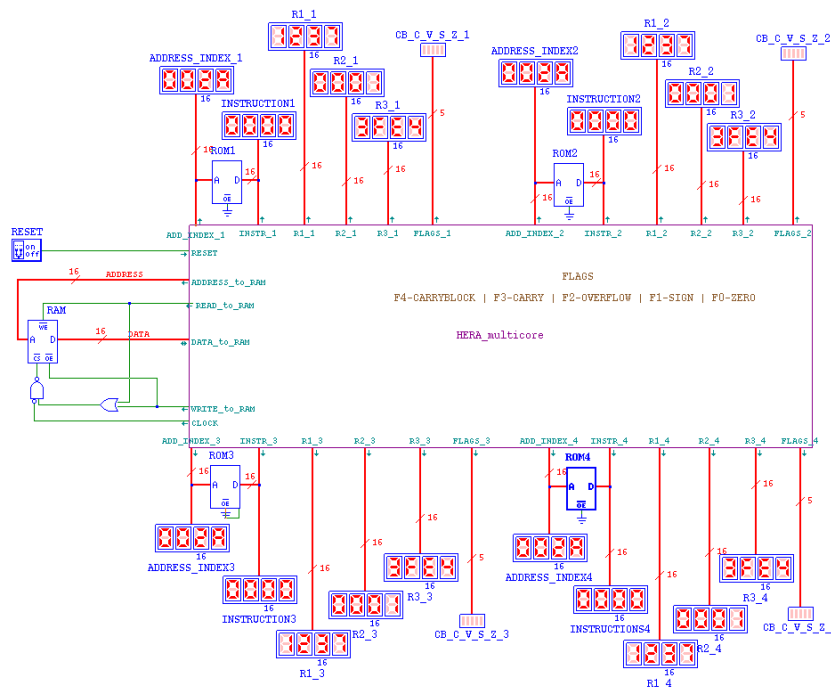


Figure 1. The HERA multi-core CPU main module.

## 2.1 Bus Arbitration

Conceptually, bus arbitration is a simple problem; don't let more than one CPU core read or write from main memory at any given time. The implementation of bus arbitration is complicated, but can be handled if the problem is broken into several sub-problems. Circuit modules can be used to encapsulate a smaller problem and are easily reproducible. This makes replicating the single-core HERA CPU simple, subsequently, four CPU cores are made, and the bus lines that used to connect to main memory, should be connected to a bus arbitration module.

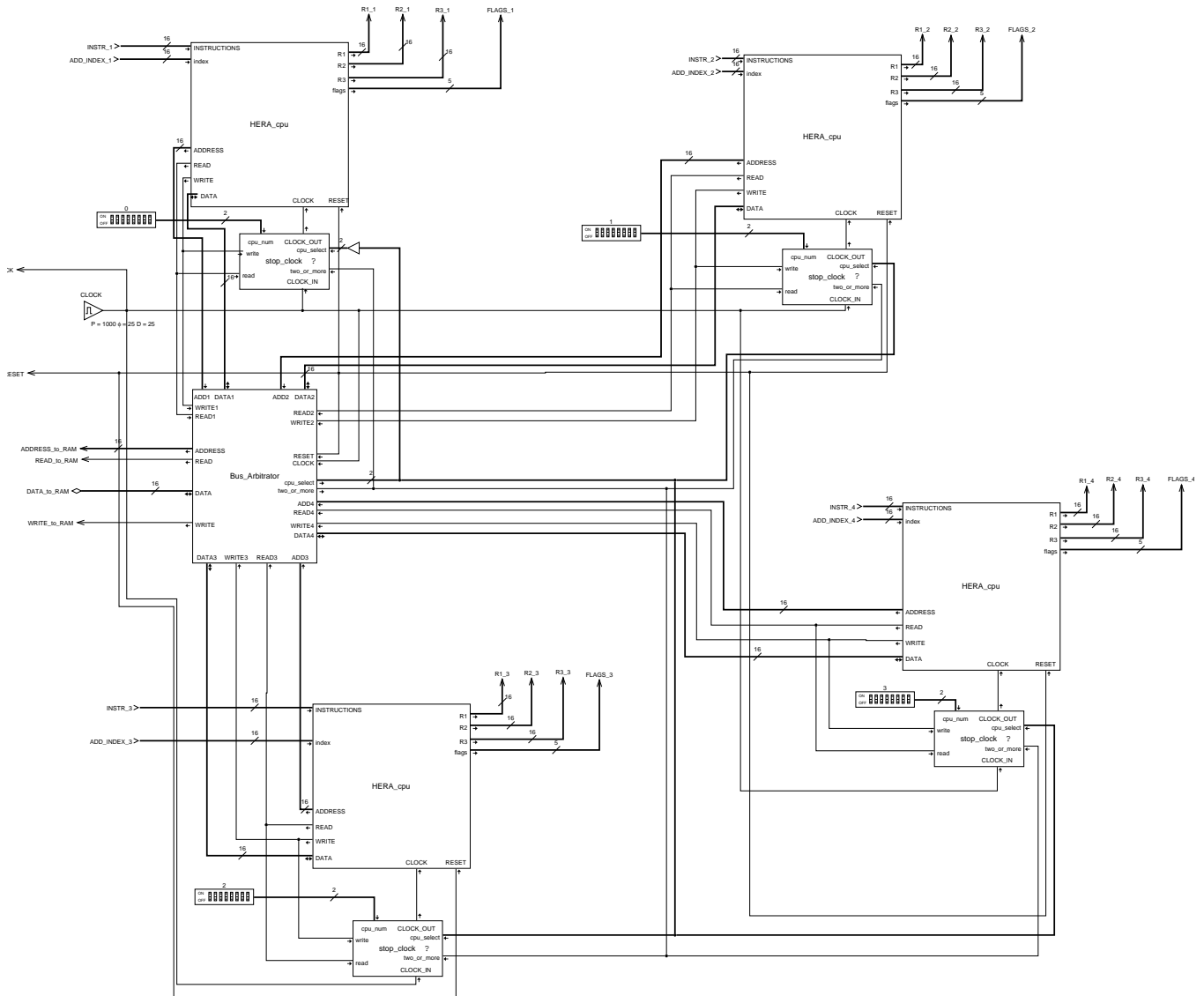


Figure 2. A HERA quad-core CPU.

### 2.1.1 Organization

Creating this bus arbitration module shown in figure 3 was tedious, but necessary to organize the inputs and outputs of the bus arbitrator. The finished module had the following inputs from each HERA CPU core: Address, Data, Read (which can be thought of as Read Enable), and Write

(Write Enable). The main outputs of the module are simply the memory outputs of a single-core HERA CPU. Nothing is changed about how memory is accessed, so connecting the bus arbitrator to the CPU is straightforward. The additional outputs are used to make decisions used in other parts of multi-core CPU easier, and will be explained in detail later.

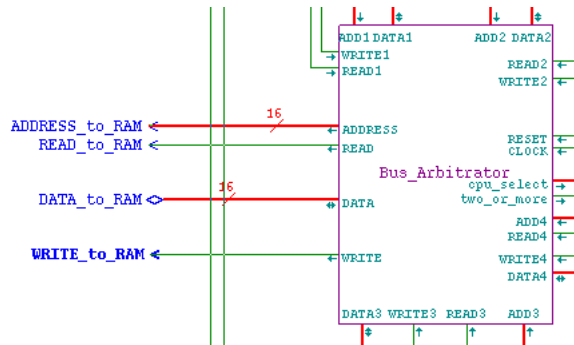


Figure 3. Organization of Bus Arbitrator module.

### 2.1.2 Identifying a Problem

How do you know if multiple CPU cores are trying to read or write to main memory? Since the bus arbitration module took Read Enable and Write Enable as inputs from each core, we can easily identify if a core wants access to memory. There are two important cases to handle: 1) either zero or one CPU core wants to access to memory, 2) two or more CPU cores want to access memory. Handling the first case goes a long way to correctly arbitrating the bus lines. The second case requires a bit of additional logic that will be used throughout the multi-core CPU, specifically identifying if two or more CPU's are making requests to access main memory. At many stages in arbitration, the same questions need to be asked: Is the CPU core I'm working with reading or writing AND are two or more CPU cores not trying to access memory? Alternatively (OR), is the CPU core I'm working with reading or writing and is it this core's turn to access memory? These questions are confusing in prose, so the figure 4 provides a more concise and accurate description of what information we need to enable different parts of the bus arbitration module.

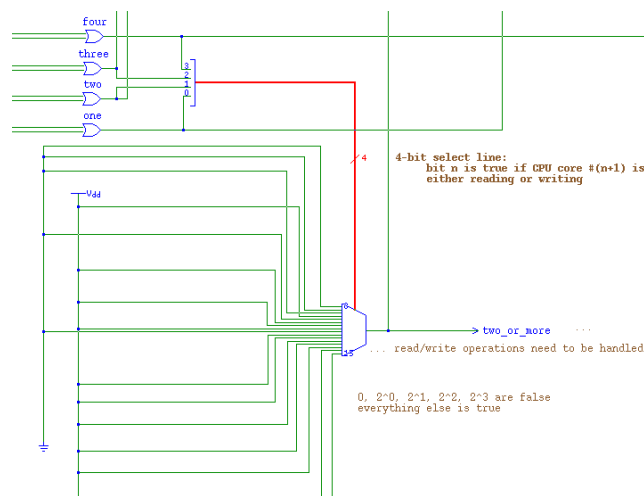


Figure 4. Circuit identifying if two or more CPU cores require access to the bus.

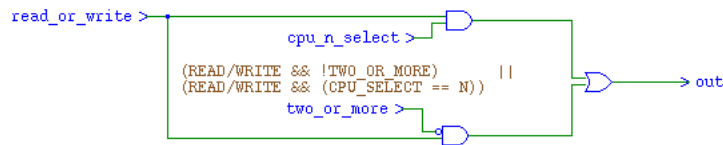


Figure 5. Bus arbitration enable logic.

### 2.1.3 Selecting Among Multiple Bi-directional Bus Lines

In order to successfully arbitrate the bus lines we must select among the correct address and data inputs. Each core has corresponding address and data lines, suppose that only CPU core #2 wants to read from memory, we need to take the address from CPU core #2, and output it from the arbitration module. For addresses this is trivial, all we need to use is a multiplexor. The difficulties come into play when we try to select among multiple bi-directional bus lines. A multiplexor is no longer sufficient because it requires a constant direction to work. The obvious remedy to this problem is a tri-state buffer, but the actual solution requires a somewhat strange arrangement of tri-state buffers. Sometimes the DATA\_N bus line (where N identifies a CPU core), is an input, other times it is an output. We need to identify these cases with the same enable logic discussed in section 2.1.2. Our goal is to allow data to flow from the correct DATA\_N bus line to DATA without allowing a DATA\_M ( $M \neq N$ ) to corrupt the bus with an undefined value. John P. Hayes' Computer Organization [Hay88] gives an example that shows the correct orientation of tri-state buffers. The actual solution to our problem is slightly different, and is shown in figure 6.

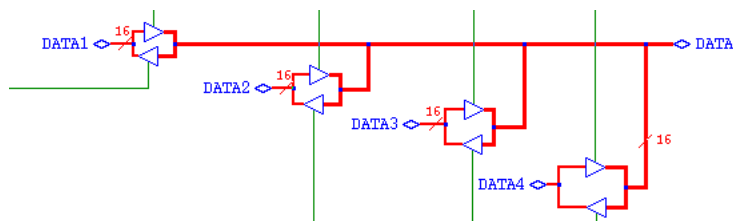
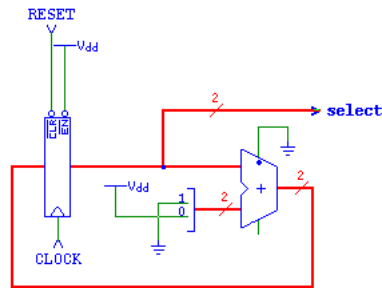


Figure 6. Orientation of tri-state buffers used to correctly select bi-directional data lines.

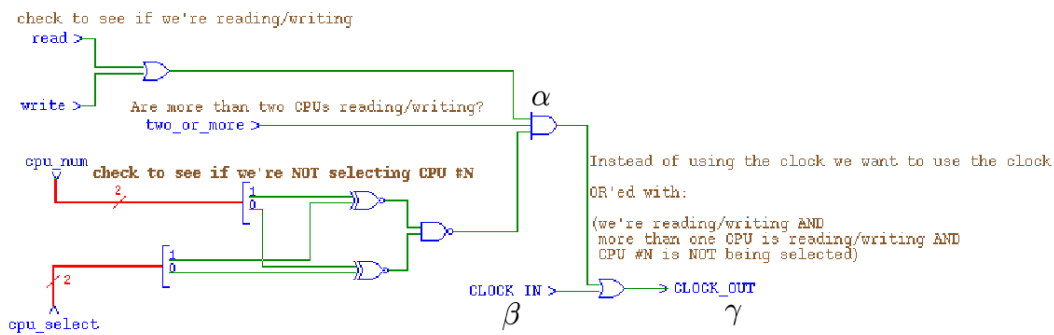
### 2.1.4 Scheduling/Fairness

Suppose three CPU cores are trying to read from main memory at the same time. Which core should go first? Which core should go second? ... A system needs to be implemented such that all CPU cores have the same priority when attempting to access memory. One idealized method of arbitration is to rotate through CPU cores on every clock tick. So, starting with the first clock tick CPU core # 1 get's priority, on the next clock tick, CPU core #2, then #3, #4, and back to #1. This is easy to implement with any two bit counter, an example is demonstrated in figure 7. Of course, if CPU cores #3 and #4 are requesting to use main memory, but CPU core #1 has priority, we need to do something to stop the progress of #3 and #4 so that they don't acquire an undefined value from memory. Essentially we need to halt the CPU core at this point, and there are a number of methods that would work in this case. If we were trying to avoid timing

issues, we could try to pause the program counter, but that requires a substantial amount of additional logic. In the case of the HERA multi-core CPU, it's easier to add logic before each CPU core clock input. We can use a multiplexor to choose between the normal clock line and the modified clock lines. To identify situations in which we want this more complicated clock signal we need to know if two or more CPU cores are requesting rights to memory (we solved this before!), and which CPU has the priority to access memory (the CPU\_select line). We choose to OR the clock with this additional information, because by leaving the clock high, we avoid unnecessary pulses inside the CPU core.



**Figure 7.** Simple counting circuit used to select which CPU core has access to main memory.

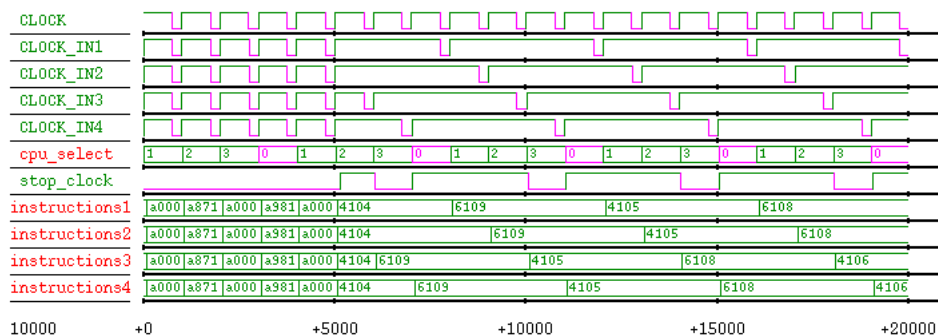


**Figure 8.** Logic added to clock lines so that CPU cores can be "halted".

### 2.1.5 Correctness

Using additional logic on the clock to halt a CPU core is an effective method of halting a CPU core, but whenever logic is added to the clock it's of utmost importance to understand where the rising edges of the clock occur. Inside the circuit simulator, users can use a virtual oscilloscope to monitor any given wire, this is particularly useful inside the stop clock module (Figure 8). The trace in Figure 9 was created by monitoring the original clock, the output of the Stop Clock module for each CPU core, the output of the CPU Select logic, and the output of the three-input AND gate inside the stop clock module. Looking at the CLOCK\_IN# lines and the instructions# lines you can see the cascading read/write priority starting with the third CPU core, and subsequent read/write operations taking place. It's clear that for each real clock cycle, the clock input to a CPU core is extended to be four times as long; the result of the OR'ing inside the stop clock module (Figure 8). The timing diagram is based on the HERA assembly language program

in Figure 10.



**Figure 9.** Oscilloscope output used as timing diagram.

```

9/ a000    // ADD
a/ a871    // ADD
b/ a000    // ADD
c/ a981    // ADD
d/ a000    // ADD
e/ 4104    // LOAD
f/ 6109    // STORE
10/ 4105   // LOAD
11/ 6108   // STORE
12/ 4106   // LOAD

```

**Figure 10.** Segment of a HERA assembly language program corresponding to the diagram in Figure 9.

The goal of the stop clock module is to halt the clock without introducing any spurious edges, and while the figures 9 and 10 display the CPU working correctly, it's not clear that it works in all cases. To show this fact, we need to show that the gate delay of inputs in figure 8 is significantly less than the length of a single clock pulse. Refer to figure 11 for a more concise description. Starting from the top left hand corner of the module, we see the read and write inputs. These inputs are pulled from the instruction decoder, and it takes an AND gate and an OR gate to identify whether a CPU core is reading or writing. The pair of gates has a delay of 14 ns when nested, which is very small compared to the length of a single clock pulse, 1000 ns. The input from the Two or More circuitry is also based on the read and write lines, additionally, it uses a multiplexor which has a gate delay of 36 ns. The number of each CPU core is done by wiring the input to be a given number, which can be done with no delay. Selecting a CPU core takes a bit more time because it involves a register that has both a setup and a hold time, that total 20 ns which combine with an adder that has another 20 ns delay. The total delay in getting the CPU select line into the stop clock module is 40 ns. After the inputs reach their destination, basic logic operations are used to modify the clock line. Looking at the three input AND gate, labeled  $\alpha$  in Figure 8, we can see that it takes

$$20 \text{ ns} = 14 \text{ ns (for read/write)} + 6 \text{ ns (for the OR gate)}$$

for the top most input,

$$50 \text{ ns} = 14 \text{ ns (for read/write)} + 36 \text{ ns (for multiplexor)}$$

for the Two Or More circuitry that is used as the middle input, and

$$52 \text{ ns} = 40 \text{ ns (for CPU select)} + 12 \text{ ns (for NOR and NAND gates)}$$

for the bottom input. The top two inputs of  $\alpha$  will run in parallel with the third input, because they are not sharing any information, so it takes a total of 52 ns for the  $\alpha$  to get all of its required information. Read/Write get there first, then Two or More, then CPU select, so depending on their respective values,  $\alpha$  may have a value prematurely. This is guaranteed not to effect the clock line, because this occurs when the clock line is high before  $\alpha$  must be true as long as the clock remains high this long. The total setup time for the clock logic is

$$66 \text{ ns} = 52 \text{ ns (Two or More)} + 8 \text{ ns (three input AND gate)} + 6 \text{ ns (final OR gate)}$$

A delay of 66 ns is extremely small compared to the clock pulse, and the clock is high during this time, so this logic prevents spurious edges on the clock line. If the clock was changed to be faster than 66 ns, this would break this circuit and many others inside the CPU, most notably the ALSU, which relies on a long clock pulse.

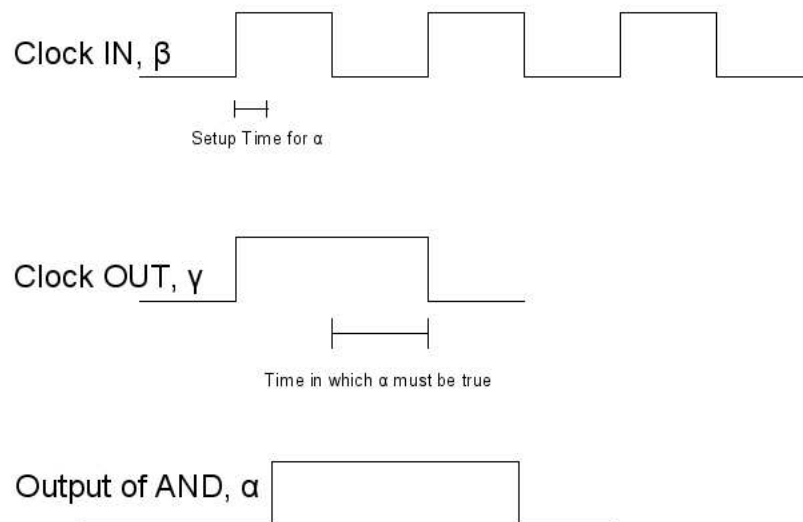


Figure 11. More specific timing diagram for Figure 8.

### 2.1.6 HERA Multi-core Design and Departures from Optimal Design

The HERA architecture is based on the idea, that each individual component should be made in the easiest way possible while still meeting the functional requirements. This is obviously not the case for real world CPUs, or even an optimally designed CPU. The most obvious HERA design decision that departs from an optimal design is the rotating pointer that defines which CPU core gets access to main memory. In the worst case, this design results in two wasted clock cycles. Suppose that CPU core #1 and CPU core #2 want to read, and cores #3 and #4 are doing arithmetic operations, but the pointer is on core #3. Both cores #1 and #2 are paused until cores #3 and #4 are done, so they do nothing until it is their turn. This inefficiency would not be acceptable for an optimal design, but since speed was never a priority in the HERA architec-

ture, a maximum delay of two clock cycles is satisfactory.

### 3 Concurrency

Bus arbitration is used to negotiate bus access when multiple read/write requests are made simultaneously, but what happens when instructions on different CPU cores refer to identical locations in memory?

#### 3.1 Example Problem

The example in Figure 12 showcases the problem mentioned above. Two CPU cores are writing to and then reading from the same locations in memory. The result is a program whose final result is not correct. CPU core #1 expects a 1 to be at location 2 in memory, but instead it sees a 2. The 2 was placed there by the program running on CPU core #2, and as a result CPU core #1 has:

$$R3 = R1 + M[R1 + 1] = 1 + 2 = 3$$

when the correct result should be:

$$R3 = R1 + M[R1 + 1] = 1 + 1 = 2,$$

but since programs running on CPU core #1 and #2 are running concurrently, the second program interferes with the first. From the example with only two CPU cores, it's clear that by sharing main memory we've created a huge problem.

<i>CPU core #1</i>	<i>CPU core #2</i>
00/ e101 // Set R1 to 1	00/ e102 // Set R1 to 2
01/ 6111 // M[R1+1] = R1	01/ a411 // filler instruction
02/ a411 // filler instruction	02/ 6101 // M[R1] = R1
03/ a511 // filler instruction	03/ 4201 // R2 = M[R1]
04/ 4211 // R2 = M[R1+1]	04/ a321 // R3 = R2 + R1
05/ a321 // R3 = R2 + R1	05/ 0000 // halt
06/ 0000 // halt	

Figure 12. Coincidental writes to the same location in memory.

#### 3.2 Lock Based Solution

If the HERA architecture had an exchange instruction, a spin lock solution could be used to solve the problem presented in Figure 12. A spin lock waits in a loop until the lock it is trying to acquire becomes available. To complete this solution, we need to encapsulate our memory write inside a loop.

**Note 1.** The *Exchange* instruction has not been implemented in HERA.

```
// Assume M[0] = 0 before parallel processing

CPU core #1                                CPU core #2
00/ e101          // Set R1 to 1            00/ e102          // Set R1 to 2
01/ e401          // Set R4 to 1            01/ e401          // Set R4 to 1
02/ EXCG(M[0], R4)// Exchange M[0], R4     02/ EXCG(M[0], R4)// Exchange M[0], R4
03/ 09FF          // BNZR(-1)              03/ 09FF          // BNZR(-1)
04/ 6111          // M[R1+1] = R1           04/ 6101          // M[R1] = R1
05/ 4211          // R2 = M[R1+1]          05/ 4201          // R2 = M[R1]
06/ 6000          // M[0] = 0 (Unlock)      06/ 6000          // M[0] = 0 (Unlock)
07/ a321          // R3 = R2 + R1          07/ a321          // R3 = R2 + R1
08/ 0000          // halt                   08/ 0000          // halt
```

**Figure 13.** By adding an Exchange instruction, we can solve the problems in Figure 12 [Tan01, Sec 2.3.2].

### 3.3 General Limits of Concurrency

As shown in figure 13, locks are an effective tool used for writing concurrent programs. In order for locking to work, one CPU core must block the other from acquiring a lock. Suppose that CPU core #2 acquired the lock first, and before it let go of the lock it ran for ten thousand instructions. This waiting time characterizes a problem with spin locks; if the problem being solved is coarse grained, then using spin locks may not be a good solution. Another option for writing concurrent programs is switching threads when two operations cannot be done simultaneously. Rather than acquiring locks, the processor can switch to another thread to avoid the problems with busy waiting. Thread switching requires a significantly higher overhead than spin locks, so it is not suited to fine grained parallelism. Thus, it is necessary to consider waiting time as compared to the time it takes to switch a thread when deciding which style of concurrent programming is to utilize.

When using locks in software, the programs become substantially more complicated as more locks are used. For this reason, locking does not scale well as a solution, and it becomes apparent that that a better method of concurrent programming is needed [ATKS07].

## 4 Extending a HERA multi-core CPU to support Transactional Memory

**Note 2.** At the time of writing TM has not been implemented on a HERA multi-core CPU.

To avoid the issues with lock-based concurrent programming we can implement hardware based transactions into the HERA architecture. With transactions implemented in HERA concurrent programming should be significantly easier for the programmer.

**Definition 3.** A *transaction* is a finite sequence of machine instructions, executed by a single process, that is both serializable and atomic. [HM93]

A transaction is used to replace the critical section, but it's important to keep in mind that since our transactional cache is small, TM only replaces a small critical section. A transaction results in one of two states, it either commits or aborts. If a transaction commits, it writes what is in the transactional cache to memory, whereas an abort discards any changes it made in the transactional cache without touching memory. A transaction is successful if no other transaction has read or written data on which it is currently operating.

## 4.1 Overview

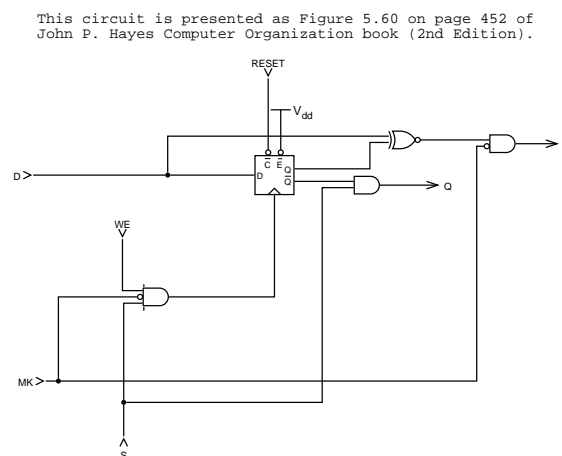
Herlihy and Moss provide a basic overview of implementing a TM system. Their system includes a fully associative transactional cache that is used when a transactional instruction is called. The modifications discussed in the original TM paper are used in a multiprocessor system, but still apply to the HERA multi-core CPU discussed in the previous sections. Herlihy and Moss summarize the changes to their architecture to be modifications to the cache-coherence policy, but since HERA doesn't use a cache, that description doesn't quite fit. HERA needs to adopt a cache coherence policy that allows the the cache lines to watch the bus and give information about it's status, in a process known as *cache snooping*.

## 4.2 Transactional Cache

In order to use hardware transactions we need a cache in which we can modify memory elements without affecting the real copy of main memory. A transactional cache is fully associative, and requires additional logic for committing and aborting transactions. Essentially, the transactional cache is a place to store “tentative writes to memory, without propagating them” to other locations before a commit [HM93]. During transactions, writes will be made to the transactional cache, while reads may take place in either the cache or main memory. If during a transaction CPU core #1 tries to read an element of memory that is in CPU core #2's transactional cache, CPU core #1 will abort it's transaction.

### 4.2.1 Associative Memory Cells

To build an associative cache, it's useful to begin with an associative memory cell and build up from there. Figure 14 provides the circuit diagram for a single bit associative memory cell. Associative memory differs from a regular memory, because it searches it's entire memory to detect if specific data is already stored somewhere.



**Figure 14.** An associative memory cell [Hay88].

### 4.2.2 Fully Associative Cache

Now, with a single bit of associative memory, you can build a fully associative cache, as described by Hayes, and shown in Figure 15. A 4x4 bit associative cache would be essentially useless in the HERA architecture, but extending this design to be a 4x16 bit or a 16x16 bit associative cache is straightforward, and would work well in a HERA CPU.

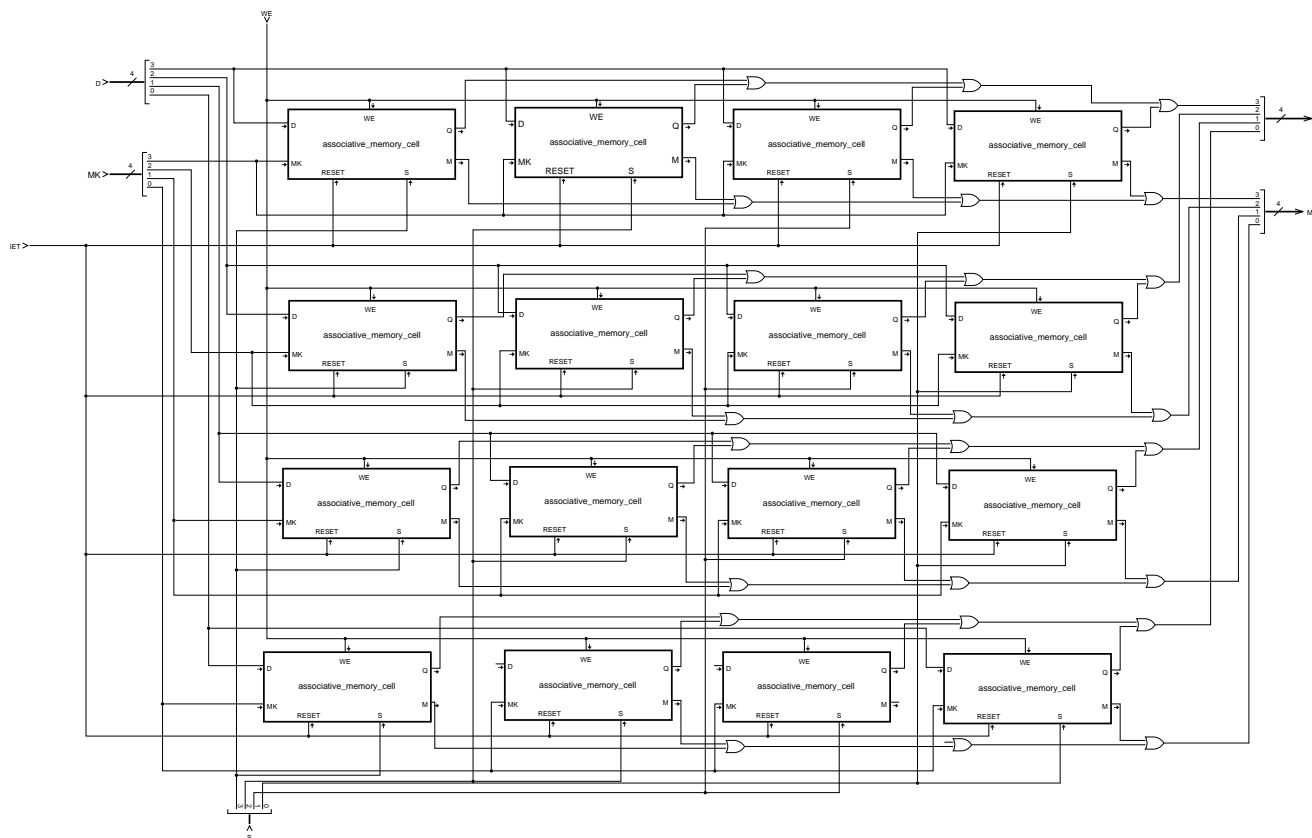


Figure 15. A 4x4 bit associative memory cell[Hay88].

### 4.2.3 Interfacing the Transactional Cache with the HERA multi-core CPU

Figure 15 would work as a fully associative cache, but it doesn't have the additional logic required for the committing and aborting a transaction. The cache itself would be placed between the HERA CPU core and main memory, preferably inside the CPU module. The transactional cache must also have the ability to snoop on the bus lines. Once in a transaction, all subsequent writes to memory go to the transactional cache rather than main memory. If the transactional cache is filled, this would be considered an error, and a reasonable solution to this problem would be branching to the end of the transaction and producing an exception. Inside of a transaction, memory reads may go to main memory, but may not touch memory cells which are being used by another transaction, otherwise the transaction will abort. If a transaction does abort, it must flush the cache without propagating any information to main memory. When a transaction completes it must commit all information in it's transactional cache to main memory. Since it's guaranteed that transactions that are committing have unique memory locations, multiple transactions can be committed concurrently. The bus arbitrator will propagate information from the cache to main memory just as it does regular writes. These writes to memory may take multiple cycles, but are still considered atomic.

### 4.3 Transactional Instructions

In order to use transactions, they need to be built into the HERA architecture. Herlihy and Moss provide six different instructions: Load-Transactional, Load-Transactional-exclusive, store-transactional, commit, abort, and validate [HM93]. Since HERA is a much simpler architecture, we can use two instructions and a flag that take the place of the six instructions.

<i>Mnemonic</i>	<i>Op. Code</i>	<i>Notes</i>
TRST	1 1 1 2	Transaction Start changes the state of the CPU from a normal process to a transactional process.
TREND	1 1 1 3	Transaction End puts the state of the CPU back to normal, resulting in a successful.

**Figure 16.** Additions to the HERA architecture.

#### 4.3.1 Implementing Transactional Instructions

With the instructions from Figure 16 now in the HERA architecture, they can be identified in the instruction decoder. When a TRST operations is identified, the CPU core that identifies it is forced into a new state. All of it's subsequent write operations must take place in the transactional cache. To make this transactional state easier to identify a new flag must also be added to the HERA architecture, but it need not be included with the five flags used in the ALSU. Instead, this new flag TFLAG can be a special 1 bit register placed inside the instruction decoder that turns on when TRST is read, and turns off when TREND is read. TREND has one possible result, commit. Commit writes all of the changes in the transactional cache out to main memory; however, a transaction can abort before it gets to the TREND instruction. When an abort occurs, the transaction will not commit it's tentative writes to memory, and it must branch back to the beginning of the TRST instruction. No transaction can attempt more than three times, and on the third failed attempt it should branch to the end of the TREND instruction and either fail, or engage the livelock prevention system.

## 5 Dining Philosophers Example

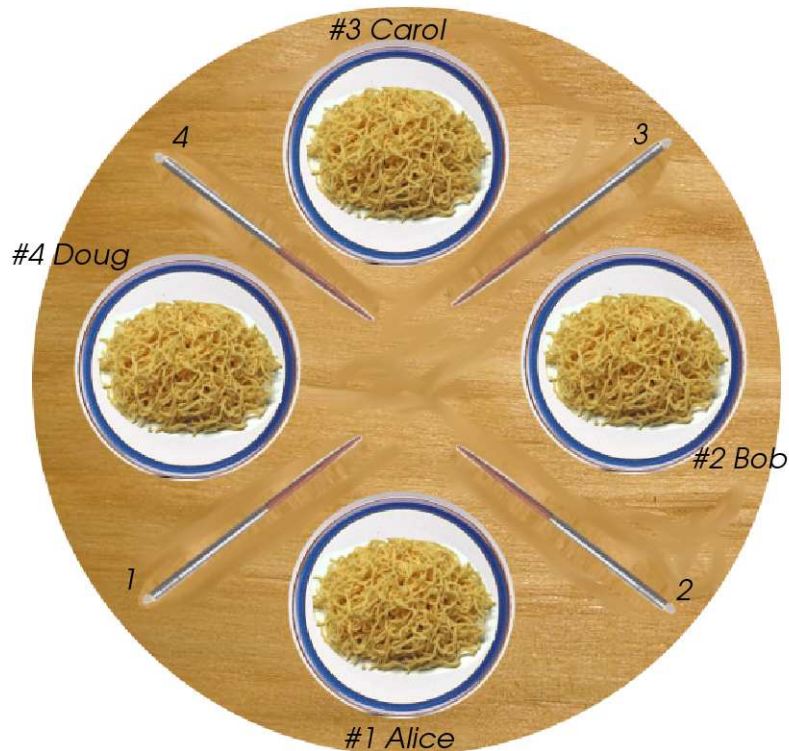
To display why TM is a better solution than using locks, we'll solve the classic dining philosophers problem using both methods. The problem was invented by Edsger Dijkstra, and it demonstrates the synchronization problems that arise from multi-processing.

### 5.1 Problem Definition

Suppose that in front of each plate of noodles in Figure 17 there was a philosopher who could do only one of two things: eat or think. In order for a philosopher to eat they need two chopsticks, but they can only pick up one chopstick at a time. There are three problems that need to be

avoided in both the lock-based and transactional solutions:

- i. If all of the philosophers simultaneously grab the left fork, and wait for the right fork, they've created *deadlock*.
- ii. If all of the philosophers simultaneously grab the left fork, try to grab the right fork, see that it is not there, put down their left fork, wait for a predetermined amount of time, and then grab the left fork entering the process again they've created *livelock*.
- iii. *Starvation* occurs if any of the philosophers are continually denied forks. Deadlock and livelock can produce starvation, but it is also possible for the philosophers to be neither deadlocked, nor livelocked and still undergo resource starvation. Consider a situation where Alice and Carol acquire both sets of chopsticks, but never let go.



**Figure 17.** Four philosophers, four plates of noodles, but only four chopsticks. How can everybody eat?

## 5.2 Lock Based Solution

There are many lock based solutions to the dining philosophers problem, the elegant solution in Figure 18 is provided in Andrew Tanenbaum's *Modern Operating Systems* 2nd Edition. As noted, there is an additional commented section of code that is used to make the transactional comparisons complete.

```

#define N          4           // # of philosophers
#define LEFT      (i+N-1)%N   // number of i's left neighbor
#define RIGHT     (i+1)%N     // number of i's right neighbor
#define THINKING  0           // philosopher is thinking
#define HUNGRY    1           // philosopher is trying to get chopsticks
#define EATING    2           // philosopher is eating
typedef int semaphore;       // semaphores are a special type of int
int state[N];               // array to keep track of everyone's state
semaphore mutex = 1;        // mutual exclusion for critical region
semaphore s[N];             // one semaphore per philosopher

// *** Additional information required for transactional solution
// Chopsticks can be in 1 of 3 states
// #define ON_TABLE      0     // chopstick is on the table
// #define IN_LEFT_HAND  1     // chopstick is in someone's left hand
// #define IN_RIGHT_HAND 2     // chopstick is in someone's right hand
// int chopstick[N]       // array to keep track of state of chopsticks

void philosopher(int i)      // i: philosopher number, from 0 to N-1
{
    while (TRUE) {          // repeat forever
        think();            // philosopher is thinking
        take_chopsticks(i); // acquire two forks or block
        eat();              // yum-yum, noodles
        put_chopsticks(i);  // put both chopsticks back on table
    }
}

void take_chopsticks(int i)  // i: philosopher number, from 0 to N-1
{
    down(&mutex);           // enter critical region
    state[i] = HUNGRY;      // record fact that philosopher i is hungry
    test(i);                // try to acquire 2 chopsticks
    up(&mutex);             // exit critical region
    down(&s[i]);             // block if chopsticks were not acquired
}

void put_chopsticks(i)      // i: philosopher number, from 0 to N-1
{
    down(&mutex);           // enter critical region
    state[i] = THINKING;    // philosopher has finished eating
    test(LEFT);             // see if left neighbor can now eat
    test(RIGHT);            // see if right neighbor can now eat
    up(&mutex);             // exit critical region
}

void test(i)                // i: philosopher number, from 0 to N-1
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
}

```

Figure 18. A solution to the dining philosophers problem [Tan01].

### 5.2.1 Discussion of Tanenbaum's Solution

Tanenbaum describes the solution as follows:

The solution presented [in Figure 18] is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, *state*, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move only into eating state if neither neighbor is eating. Philosopher *i*'s neighbors are defined by the macros *LEFT* and *RIGHT*. In other words, if *i* is 2, *LEFT* is 1 and *RIGHT* is 3.

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure *philosopher*, and *test* are ordinary procedures and not separate processes. [Tan01]

The `down()` and `up()` operations on the semaphores are atomic actions, equivalent to the spin lock implemented with an exchange operation in Figure 13. While it's true that the philosophers can eat in parallel, it is not true that they can use `test()` in parallel. Since the `up()` operation is atomic, only one philosopher can access it at any given time for a given semaphore. More importantly, the main constraint with this algorithm is that a single semaphore is used to control access to the critical regions in `put_chopsticks()` and `take_chopsticks()`. The semaphore `mutex` can only be acquired by one philosopher at a time, it successfully allows only one philosopher into the critical region, but it also creates more overhead, and some unneeded waiting when philosophers who aren't neighbors lock each other from chopsticks. Each `up()` and `down()` operation on a semaphore translates into multiple machine instructions even if they fail. With transactional memory implemented we can avoid some of that overhead, with more concise code that is fully parallelizable.

## 5.3 Transaction Based Solution

The transactional solution to the dining philosophers problem presented in Figure 19 is clearly much easier for a programmer. The details of the transactional implementation, specifically how a transaction fails, are not as easy to understand. From a higher level view, these details are irrelevant, because they are taken care of inside the hardware. The programmer does not have to solve these problems to run correct and efficient code, and this makes transactional programming significantly easier.

```

#define N          4           // # of philosophers
#define LEFT      (i+N-1)%N   // number of i's left neighbor
#define RIGHT     (i+1)%N    // number of i's right neighbor
#define THINKING  0           // philosopher is thinking
#define HUNGRY    1           // philosopher is trying to get chopsticks
#define EATING    2           // philosopher is eating
#define ON_TABLE  0           // chopstick is on the table
#define IN_LEFT_HAND  1       // chopstick is in someone's left hand
#define IN_RIGHT_HAND 2       // chopstick is in someone's right hand
int chopstick[N]           // array to keep track of state of chopsticks

void philosopher(int i)     // i: philosopher number, from 0 to N-1
{
    while (TRUE) {         // repeat forever
        think();           // philosopher is thinking
        Transaction {     // Begin Transaction
            take_chopsticks(i); // acquire two forks or block
            eat();           // yum-yum, noodles
            put_chopsticks(i); // put both chopsticks back on table
        }                  // End Transaction
    }
}

void take_chopsticks(i)     // i: philosopher number, from 0 to N-1
{
    state[i] = HUNGRY;      // record fact that philosopher i is hungry
    chopstick[i] = IN_LEFT_HAND; // grab chopstick for left hand
    chopstick[(i+1)%N] = IN_RIGHT_HAND; // grab chopstick for right hand
}

void put_chopsticks(i)     // i: philosopher number, from 0 to N-1
{
    state[i] = THINKING;    // philosopher has finished eating
    chopstick[i] = ON_TABLE; // put chopstick i on table
    chopstick[(i+1)%N] = ON_TABLE; // put chopstick (i+1)%N on table
}

```

Figure 19. Pseudo-code for a transactional solution.

### 5.3.1 Discussion of Transactional Solution

This solution in Figure 19 looks as if it's glazing over the details of concurrent program, but the simplicity of the solution is the essence of transaction-based programming. Since there are no issues with acquiring locks inside we simply encapsulate the calls to `take_chopsticks()` and `put_chopsticks()` inside of a transaction. The chopsticks are labeled so that if a philosopher acquires less than two chopsticks and attempts to eat, the transaction can abort. Since the transaction is inside of an infinite loop, it's clear that after aborting it will simply retry (even if we don't define the exact situations that cause a transaction to abort, like Herlihy and Moss). One potential problem with our transaction implementation is that if two transactions begin simulta-

neously, we have to choose among the different processes to decide which gets to go. For our purposes we can assume that if different processes begin transactions simultaneously, ties are broken randomly. This is fair, and most importantly, allows the program to progress. The `test()` function from Tanenbaum's solution is no longer needed. If a transaction is aborted, and retries, there is a possibility for livelock if the aborted transaction can then cause another transaction to abort. Like Herlihy and Moss, we leave the details of resolving this undefined.

More specifically, if you consider if the philosophers Bob and Carol trying to acquire forks at the same time. The transactional solution to this is only slightly better than the locking approach, because a transaction aborts once it sees that either Bob or Carol has acquired a chopstick, while the lock-based solution has to block with a semaphore and release the critical region. Depending on how the first fork was acquired, the other philosopher will be denied, so that at least one philosopher can eat, preventing starvation. The extra instructions from the locking operations are minimal, so the transactional solution is just slightly more efficient.

Now, if Bob and Doug try to acquire forks at the same time, they will succeed in parallel when using the transactional approach, with no need to pause while the philosopher across from them enters or exits the critical region locked with `mutex` in Figure 18. As in the lock-based approach, philosophers can eat in parallel when inside of a transaction. Another benefit from transactions is that the philosophers can put down the chopsticks in parallel, without having to acquire the lock `mutex` before entering the critical region. So, the transactional solution to the dining philosophers problem is both more concise and more efficient than Tanenbaum's lock based solution.

## 6 Conclusions

The extension from a single core processor to a multi-core processor is not trivial, but the changes to the overall architecture are minimal. With modular circuitry, a bus arbitrator can be built to negotiate among multiple CPU cores so that only one core has access to main memory. With an increase in parallelism inside the CPU, the programmer must deal with the concurrent design. Using exchange instructions results in correct, but not necessarily efficient programs. Locking does not scale well as a solution, and it makes programming more difficult. From the perspective of the programmer, transactions are ideal, because they remove some of the more difficult subtleties that come with lock-based programming. A transactional memory implementation provides the programmer with the ability to write highly concurrent programs with minimal effort, by encapsulating difficult code inside of transactions. Implementing TM is far from trivial, in fact it requires fundamental changes to the cache architecture. Specifically for HERA a small fully associative transactional cache must be added, with instructions capable of utilizing the cache for manipulating data without affecting the data in main memory. With TM implemented it's clear that solving concurrent problems has been made significantly easier. In addition to making programs easier, Herlihy and Moss claim that TM provides a substantial performance increase provided the data sets were relatively small, and the duration of the transactions were short.

## Bibliography

- [App02] Andrew W. Appel. *Modern Compiler Implementation in C, 2nd edition*. Cambridge University Press, 2002.
- [ATKS07] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. *Queue*, 4(10):24–33, 2007.
- [BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [DFL06] Peter Damron, Alexandra Fedorova, and Yossi Lev. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM Press.
- [Hay88] John P. Hayes. *Computer architecture and organization; (2nd ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1988.
- [HLM06] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 253–262, New York, NY, USA, 2006. ACM Press.
- [HM93] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.
- [Moi05] Mark Moir. Hybrid transactional memory, Jul 2005. Unpublished manuscript.
- [MS04] Virendra J. Marathe and Michael L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept., Jun 2004.
- [PH05] David A. Patterson and John L. Hennessy. *Computer Architecture*. Morgan-Kaufmann, San Francisco, CA, USA, third edition, 2005.
- [SGG00] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. *Applied operating system concepts*. John Wiley & Sons, Inc., New York, NY, USA, 2000.
- [ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.
- [Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
- [Won06] David G. Wonnacott. The Haverford Educational RISC Architecture (HERA) Version 2.2.2, 2006.