Final Report - Dual Core pipelined processor with caches

ECE437 : Computer Design and Prototyping

TA : Jason Holmes

April 15, 2013

Alex van Almelo and Vineeth Harikumar

Executivlle Overview

For this lab section of ECE 437, our group was tasked to take our previous design of a single core processor that implemented a pipelined design and improve on its performance. Previously, we determined that a single cycle processor without pipelining was more efficient than one with the pipeline design implemented. By adding both caches and multicore operation into our design, we have increased the performance and completed the final requirement for this lab section of the class. Our multicore design was implemented with caches, and works in source code, in synthesis, and in hardware as well.

In the rest of our report, there will be information on the design of our pipelined processor with caches. After this the design of the multi core processor will also be outlined. Finally answers for the included processor debugging problem, results and statistics of our processor design, and a conclusion of our project for this course is also included in this report.

Processor Design

To add caches to the processor, first the icache was added to the pipeline in between the instruction fetch signals and the memory arbiter (Appendix A Figure 1.1). The icache used was the design provided by the course staff. Unfortunately integrating this into the existing pipeline was painfully hard. This turned out to be due to the fact that the timing of the signals that the icache expected for its correct functioning were different than what the existing design had in place. So after a few days of debugging and correcting the signals going from the instruction fetch stage to the icache, between the icache and the memory arbiter and ensuring that every signal was asserted at the right clock edge the icache was was finally integrated into the pipelined processor.

After this work started on designing the dcache from ground up. The total data capacity of the dcache is 256 bytes and it is a write back. It is also two way associative and contains two word cache blocks. The overall dcache design includes a dcache controller, which is a state machine and also includes two dcache RAM blocks which hold the data, each of which are conveniently named WAY0 and WAY1 (Appendix A Figure 1.2). The RAM block contains a standard logic vector that can hold 16 cache blocks, each containing 2 words(64 bits in all). It also contains a 16 bit standard logic vector to keep a track of the validity of each cache block, another 16 bit standard logic vector to keep a track of whether the data block is dirty or has had its value changed by a processor operation. It also contains a standard logic vector that can hold 16 standard logic vectors, each being 25 bits wide that hold the tag. The tag corresponds to bits 31 downto 7 of the data address. The dcache also contains a dcache controller, which is a mealy state machine that controls the flow of data between the dcache RAM blocks, the memory arbiter and the pipeline itself. This state machine has the following state : IDLE, WRITE_WORD0, WRITE_WORD1, FETCH_WORD0, FETCH_WORD1, DUMP_WAY0_WORD0, DUMP_WAY0_WORD1, DUMP_WAY1_WORD0 and DUMP_WAY1_WORD1. Anytime there’s a cache hit or the dcache isn’t responding to a load or store instruction from the processor, it remains in the IDLE state. If there is a load or a store instruction and it happens to be a cache miss, it moves to the FETCH_WORD states and starts fetching both word 0 and word 1 that belong to that cache block and halt the pipeline until both of them are fetched. The dcache also implements a LRU(Least recently used) policy. Each time it sees a load or store instruction from the processor, it decides which one of the two dcache RAM blocks contains or holds the data corresponding to the address and flips the LRU bit. This allows the dcache to decide which RAM block the next incoming data will be stored to, incase of a conflict. Since the dcache implements a write back policy, data is only kicked out of the RAM blocks if the data in the RAM block being written to has its dirty bit asserted. Each time a cache block has to be written back to memory it goes through two states, WRITE_WORD0 and WRITE_WORD1 during which both word 0 and word 1 in the cache block is written to memory respectively. Finally the state machine also implements a dump logic. This is required because due to the write back nature of the dcache, when the processor halts dirty cache blocks can be present in the RAM blocks. During the dumping logic the state machine makes each of the RAM blocks output both words in each of their 16 cache blocks one at a time, recreates their corresponding address by concatenating the stored tag, the index where the data is stored and word offset and writes each of them back to their respective address locations.

Once the two caches were added to the pipelined processor, work began on the dual core processor. Two instances of the single core processor were instantiated as a base design for the dual core processor. Each of these cores were named core0 and core1. Core0’s program counter begins executing instructions at address 0x0000 whereas core1’s program counter begins executing instructions at address 0x0200 .At first it was decided that one top level arbiter would be the only arbiter that existed in the entire design. It would be the job of this top level arbiter to allow each of the four caches between the two cores access to memory. The dcache of core0 was given the highest priority, followed by the dcache of core1, following by the icache of core0 following by the icache of core1. This involved having to remove the memory arbiter from each of the individual cores and send all the dcache and icache signals out of each of the individual cores. This design decision turned out to be the worst decision during the course of building the entire processor because of the sheer number of problems it introduced. This was because the memory arbiter in each of the individual cores was only only responsible for controlling the data flow between the icache and the dcache, but it was partly responsible for the stalling logic in the pipeline itself. So removing the arbiter from the cores caused the pipelines to behave erratically and causes it to never halt or was executing instructions incorrectly.

It was decided that instead of spending the countless hours fixing all the above problems it would be easier to scrap the design and start from scratch on an alternative design path that would introduce very few, if not zero bugs and errors. This involved building a two tiered memory arbiter system by keeping the current design of the core intact and not removing the arbiter from the core. The outputs from each of the cores would then be fed into the top level memory arbiter which would simply decide whether to service core0 or core1(Appendix A Figure 1.3). This worked out well as a design choice because the current memory arbiter in the core already handles requests for access to data from the dcache and the icache and gives preference to the dcache and sends the request out of the core itself. The only additions to the individual cores were LL/SC flag that are generated by the controller block anytime a LL or SC instruction is decoded in the decode stage. A link register was also added to the dcache controller to store the address that is associated with the LL instruction in the memory stage of the pipeline. Logic was also added to invalidate the link register on a successful SC operation. Finally, it was found that a SC instruction followed by a SW instruction from the same register was a data hazard because it is inherently the same as a LW followed by a use. To fix this problem, the hazard detection unit was modified to add a NOP instruction in between a SC followed by a SW to the same register. After this the top level memory arbiter was designed, which would simply have to decide whether to allow core0 or core1 access to memory. To prevent a situation where it gives continuous access to only one of the cores, a system similar to the least recently used policy was implemented where it alternates between giving each of the two cores access to memory. In the single core design, the RAM state signal was fed directly to the core, but in the dual core design since both cores could be requesting for data at the same time, the top level memory arbiter needed to fake the RAM state going to each of the cores. This was done in a way that would make one of the cores think the RAM is busy fetching its data while infact it is actually busy fetching the data for the other core. With very few signals additions to this top level arbiter, the initial dual core processor was able to function effortlessly and execute parallelized programs that operated on exclusive memory address locations and contained LL/SC instructions. Finally signals were added between the two cores' dcaches so that SC instructions could invalidate link registers in the other cores dcache every time its own link register was invalided if the addresses in the link registers matched.

After the success with the initial dual core design, work began on the design of the only remaining part of the dual core design - the cache coherence controller. After a lot of thought it seemed like there was no reason to separate the top level arbiter and the cache coherence controller(coco) and so the coco-arbiter block was born. First the snooping logic was built into the coco-arbiter block. Two signals were added that were fed into each of the two cores and these originated from the coco-arbiter block. One of these signals contained the address to be snooped and the other contained a snoop flag. Inside each of the cores, a snoop state was added to the dcache controller. This was done so that anytime the other core needed to snoop, the current core would halt its processor for a single clock cycle and allow the other core access to its dcache to snoop for the required cache block. It soon became apparent that this would not only be a pretty big performance hit but also be complicated to design in code. So this design was scrapped and it was decided that the coco-arbiter block would actively snoop inside the core while it was running without interrupting the cores regular operation.The only change to each of the cores was the addition of an extra input read port and an output port to the dcache that was reserved only to allow the coco-arbiter block to snoop. This design change made more logical sense to implement because the snoop states were added only in the state machines of the coco-arbiter to allow one core to snoop into the other cores dcache. The coco-arbiter states were modified so that anytime one of the cores requested data, the coco portion of the block would first spend one clock cycle snooping in the other core to check if the data existed there. If the core reports back with a snoop hit, then the data that is read is fed back to the core requesting the data. If the core reports back with a snoop miss then the arbiter portion of the coco-arbiter block takes over and goes to memory to fetch the word being requested

Once the snooping states worked and each of the cores were able to retrieve data from the other core if it was a snoop hit design of the cache coherence began. For this the MSI(Modified, Shared, Invalid) protocol (Appendix A Figure 1.5) was required to be implemented to maintain cache coherence. The MSI protocol requires that data in the cache be in one of the three MSI states at any given point of time. If the data is in the modified state then that means that the data is both valid and dirty, if the data is in the shared state then that means the data is only valid and unchanged and if the data is in the invalid state that means the data in that cache block is neither dirty nor valid. By design, the transitions from invalid to shared and shared to modified already exist in the dcache. The only additions that needed to be made were transitions from the modified to the invalid and shared states and from the shared state to the invalid state. To achieve this the input signals to the dcache from the memory stage of the pipeline in each of the cores had to be sent out of the core and to the coco-arbiter block. The coco-arbiter block was designed to continuously check these signals and interpret whether the data is moving to the modified state in one core. In such a case it signals the other core to check if that address block exists in its own cache and if it does then tells it to invalid its copy since only one dirty copy of the same data can exist in either caches at any given one point of time so as to maintain coherence. Our design was also modified to not do write backs to memory anytime there was a snoop hit and the data block was dirty and transferred to the other core. This was done so as to save the RAM latency*2 number of clock cycles it would take to write back the data to memory. Instead if a data block is dirty and valid in one core and the other core requests the data, the data block is instantly transferred to the core requesting it and instantly invalidated in the core the data originated from. This could be seen as a disadvantage when the processor is executing programs where theres one store followed by a lot of loads, but on the other hand in programs where stores follow loads, this would he a huge spike in performance time.

Processor Debugging

The output word at address 0x0090 in the memout is DEEF, whereas if the dual core processor design worked correctly, the output would've been DEAD.

There are two possible reasons why the cache hierarchy could've caused this incorrect output. The first reason is that the SC instruction didn't invalidate the link register in the dcache its own core and the other core on a successful SC instruction. The second reason could be that the snooping and invalidation logic wasn't implemented correctly.

For the first reason, if the SC instruction doesn't correctly invalidate the link register in the dcache of both the cores then both the cores could call JAL LOCK at the same time, and go through the LL and SC instructions. Both the SC instructions would then execute on both cores and since neither one is correctly invalidating the others link register, they will both succeed. This essentially means that both cores will think they each have the lock at the same time. They will then both load the word 0xBEEF at location 'res' by sending the load word request through the top level arbiter and load them back into their respective dcache cache blocks. After this they will both perform the addiu instruction and store the result back into the dcache. Assuming core0 is the last one to store the cache block containing the result at address 0x0090, then the final result written to memory will be 0xDEEF thus replicating the output in memout.hex. To distinguish and look for this error in the waveforms, one would simply have to look at the link register in both the cores and check to see if the SC instruction in each of the cores successfully invalidates the link register on both cores whenever the instruction succeeds. If the invalidation doesn't happen on both cores at the same time then thats the error.

For the second reason, whenever one core needs to LW at address 0x0090 it first needs to snoop and check if the word already exists in a cache block in the dcache of the other core. If core0 reaches the LW $t0, 0($t2), it should first trigger the snoop logic in core1. If core1 contains a dirty copy of the cache block it should return the dirty cache block to core0. But if the snoop logic is implemented incorrectly and always returns a snoop miss, then the dirty block may never be sent to core0. This would result in core0 having to go to memory and fetch the word from memory. Thus instead of carrying out the operations on the updated cache block, it would operate on the word in memory which is 0xBEEF. This would result in it storing 0xDEEF to memory instead of 0xDEAD. To distinguish this on the waveforms one would simply have to check the cache blocks in each of the dcaches of the cores when the other core is snooping into its dcache. If the cache block exists in the dcache, and the dirty bit of the cache block is asserted, then the dcache needs to send out the dirty cache block to the core requesting for the cache block and then invalidate the cache block in its own dcache once it has been transferred to the other core. If the cache block exists and it dirty and the snooping logic still returns a snoop miss, then that is a design flaw because the core requesting the data will then go to memory to fetch the outdated word instead of the updated word present in the other dcache.

A very simple test that could be done to distinguish which one of the two bugs are actually causing the problem is by writing a test case that just implements the LOCK subroutine by using the LL/SC combination of instructions. If both the cores succeed with the LOCK subroutine at the same time then clearly the SC invalidation has been incorrectly implemented. On the other hand to test if the second bug is the one causing the problem, a simple test case where core0 writes to word 0 of a certain cache block and core1 writes to word 1 of the same cache block. Then the final result written to memory can be checked and if the result doesn’t reflect changes to both words, then that shows that each of the cores encountered snoop misses and hence went to memory to fetch both the words for their respective operations.

Results

Below is a table of information and calculations of our processor, including estimated clocking speed, FPGA resources, average CPI, and latency of our processor design. All programs were tested using the estimated clock period of 66 ns, as pointed out by our critical path and log files.

Table 1 - Multicore calculations

Estimated clocking speed	15.36 MHz
Estimated clock period	65.124 ns
FPGA Resources	21,238 logic elements (64%) (17,162 combinational functions, 10,731 dedicated logic registers)
Average CPI*	3.2
Average Latency of one instruction	198 ns

*Number calculated with dual.mergeSort.asm and example.asm

Table 2 - Simulation results

Assembly file	Clock cycles	Instruction count	CPI
dual.mergeSort.asm	15,326	14,544	1.05
example.asm	819	153	5.35
dual.mergeSort2.asm	17936	19901	0.901

*Code for each of the three files are attached in Appendix B

As pointed out above in our table, running a multicore processor on a program with a very few number of instructions is inefficient. Example.asm runs at a CPI of 5.35, way above the CPI from our midterm report and pipelined single cycle processor. However once the number of instructions rises greatly, the CPI shows how efficient the multicore can be, and drops the CPI close to 1, if not below it.

For our processor's critical path, both cores have a critical path starting from the top level of core and end inside a register, located in the pipeline registers. Core 0 has the critical path end in the IF/ID pipeline register, while Core 1 has its critical path end in the EX/MEM pipeline register. The largest critical paths lie in the actual cores themselves, where the critical paths inside the memory and memory controllers are almost half the speed, causing the limiting factor for our CPU's clock speed to be inside the cores. While this does limit the CPU speed, this is an acceptable critical path, since with the design re-work of our processor would cause an increase in complexity, therefore, increasing the chance of a long critical path.

Conclusions

Given the task to design and implement a multi core processor with caching and pipelining, our group was successful in completing the task. With the base of a single cycle processor, adding on the pipeline design caused our processor to drop in efficiency, due to the amount of time spent loading instructions, then tossing them out due to a branch, stall, jump, etc. After implementing caching, our processor improved on efficiency, and in programs with small loops (such as searching or sorting) our cache would have a large percentage of data hits, making our programs run much more efficiently than with just pipelining implemented.

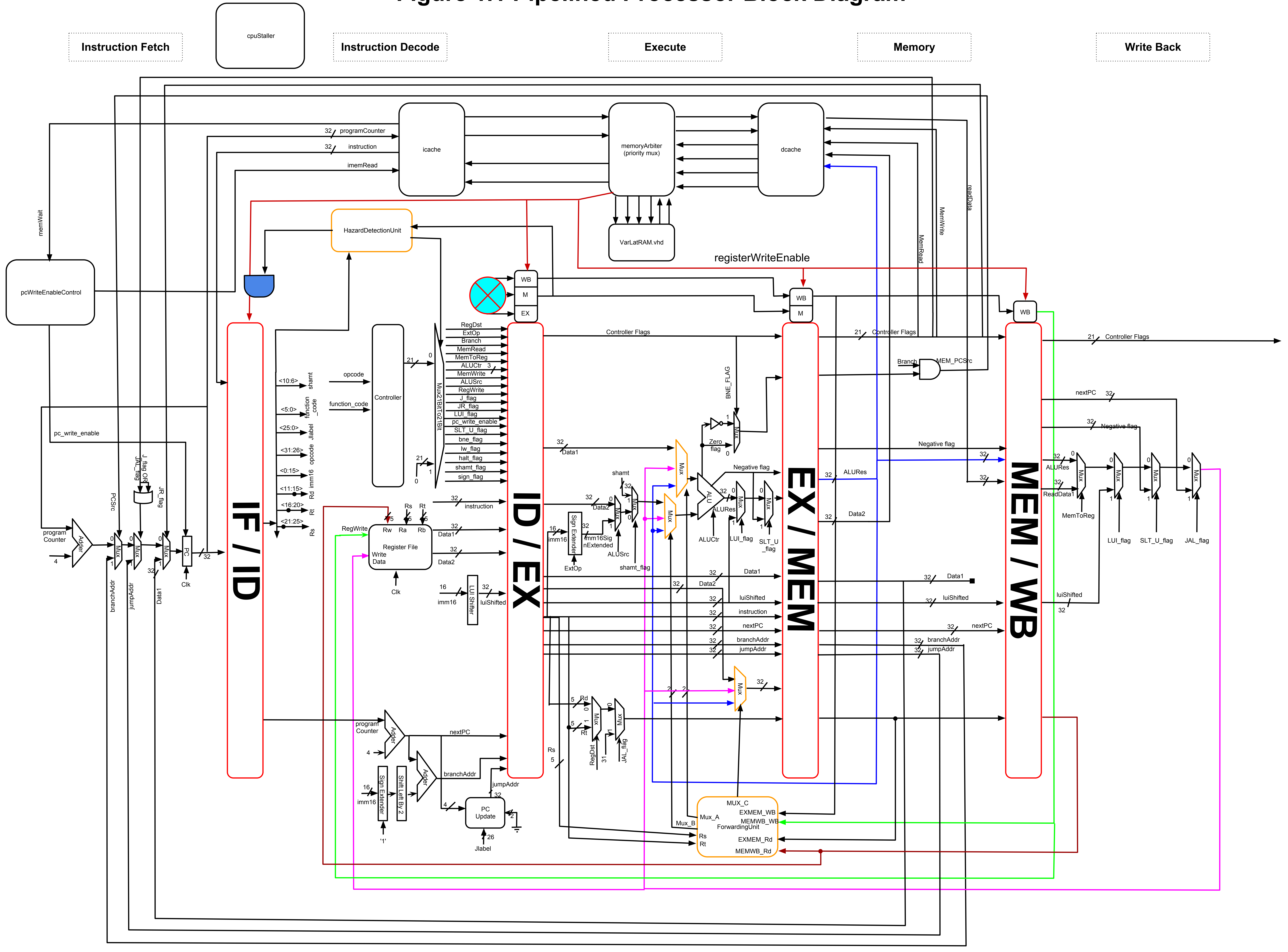
After completing caching, we worked to make a multicore processor, adding a few more controllers and logic lines to allow both cores to access memory and snoop into different data caches. We got multicore to eventually work with source, synthesis, and hardware implementations, and as pointed out in our Results section, with complex programs that have a high number of instructions, a very efficient multicore processor.

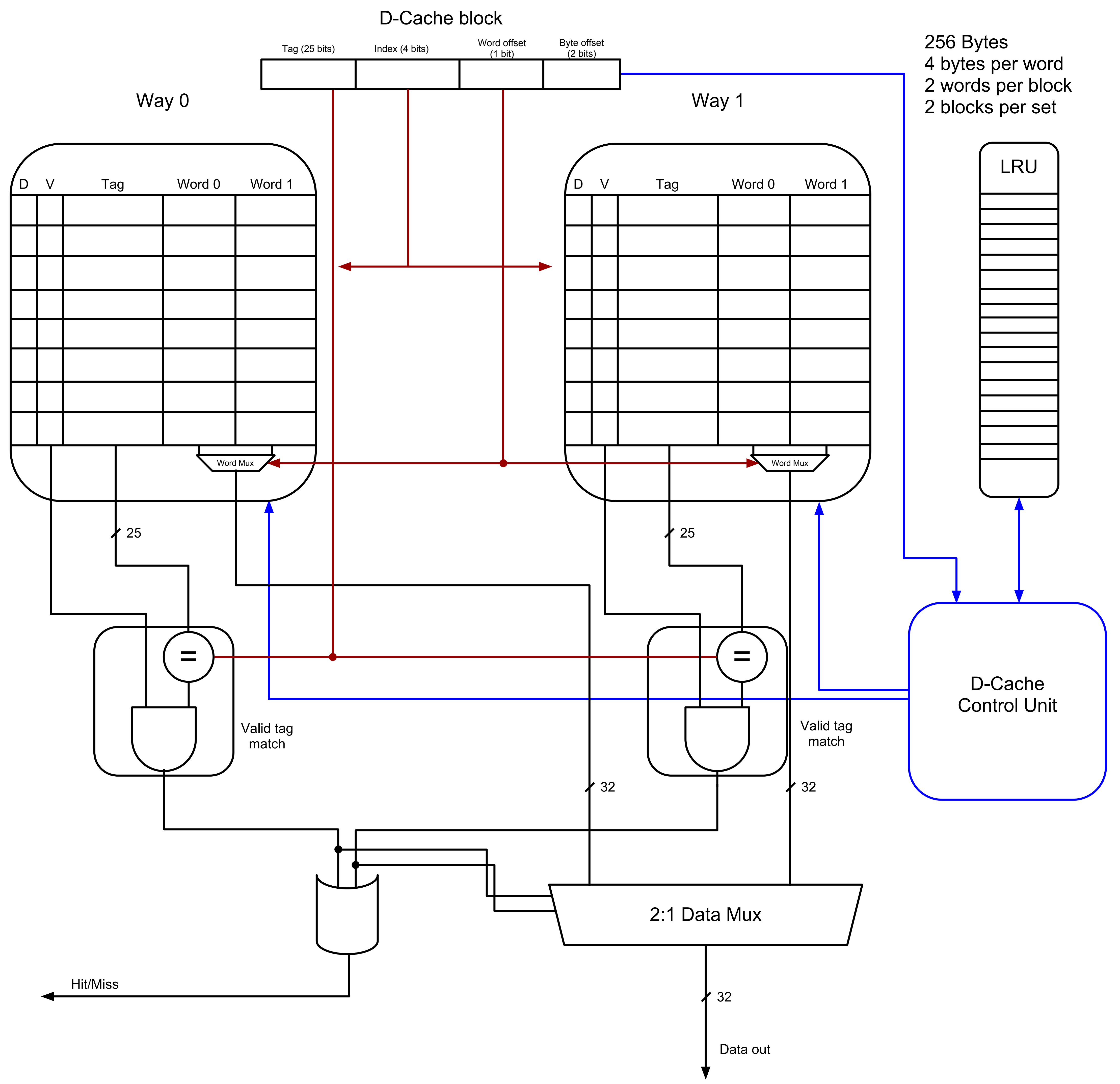
The advantage of a multicore processor with data caching is that the workload can be divided up between the cores, increasing the overall processor throughput greatly. However a disadvantage is that if a program is not very complex or long, the efficiency suffers greatly. Also, the critical path of our processor was significantly higher than our previous report on our pipelined single cycle processor, which is a limiting factor on how fast our processor is allowed to run.

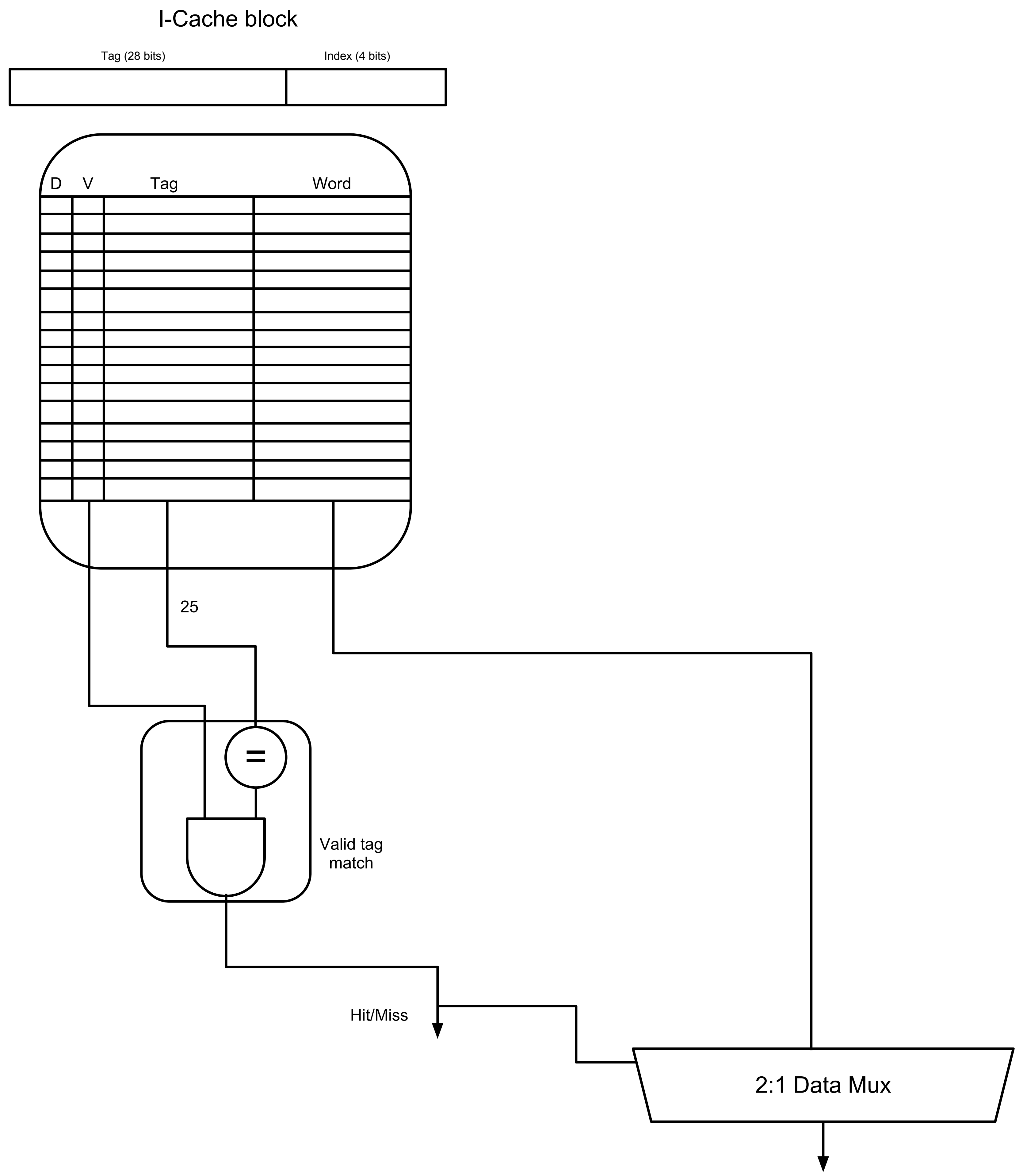
With data and instruction caching, the processor has to spend some time going through the caches and storing and reading data. While this can slow down the processor, having caches pays off when a processor is running on a set of data with high locality in memory. Things such as sorting and frequent loops usually use small sets of memory and information, and with caching, this reduces the amount of time the processor uses going in and out of the memory, causing a steep rise in efficiency

Appendix A

Figure 1.1 Pipelined Processor Block Diagram




D-Cache block
Tag (25 bits)
Index (4 bits)
Word offset (1 bit)
Byte offset (2 bits)
256 Bytes
4 bytes per word
2 words per block
2 blocks per set
Way 0
Way 1
D
V
Tag
Word 0
Word 1
Word Mux
LRU
25
32
Valid tag match
D-Cache Control Unit
2:1 Data Mux
Hit/Miss
Data out


I-Cache block
Tag (28 bits)
Index (4 bits)
D
V
Tag
Word
25
Valid tag match
Hit/Miss
2:1 Data Mux

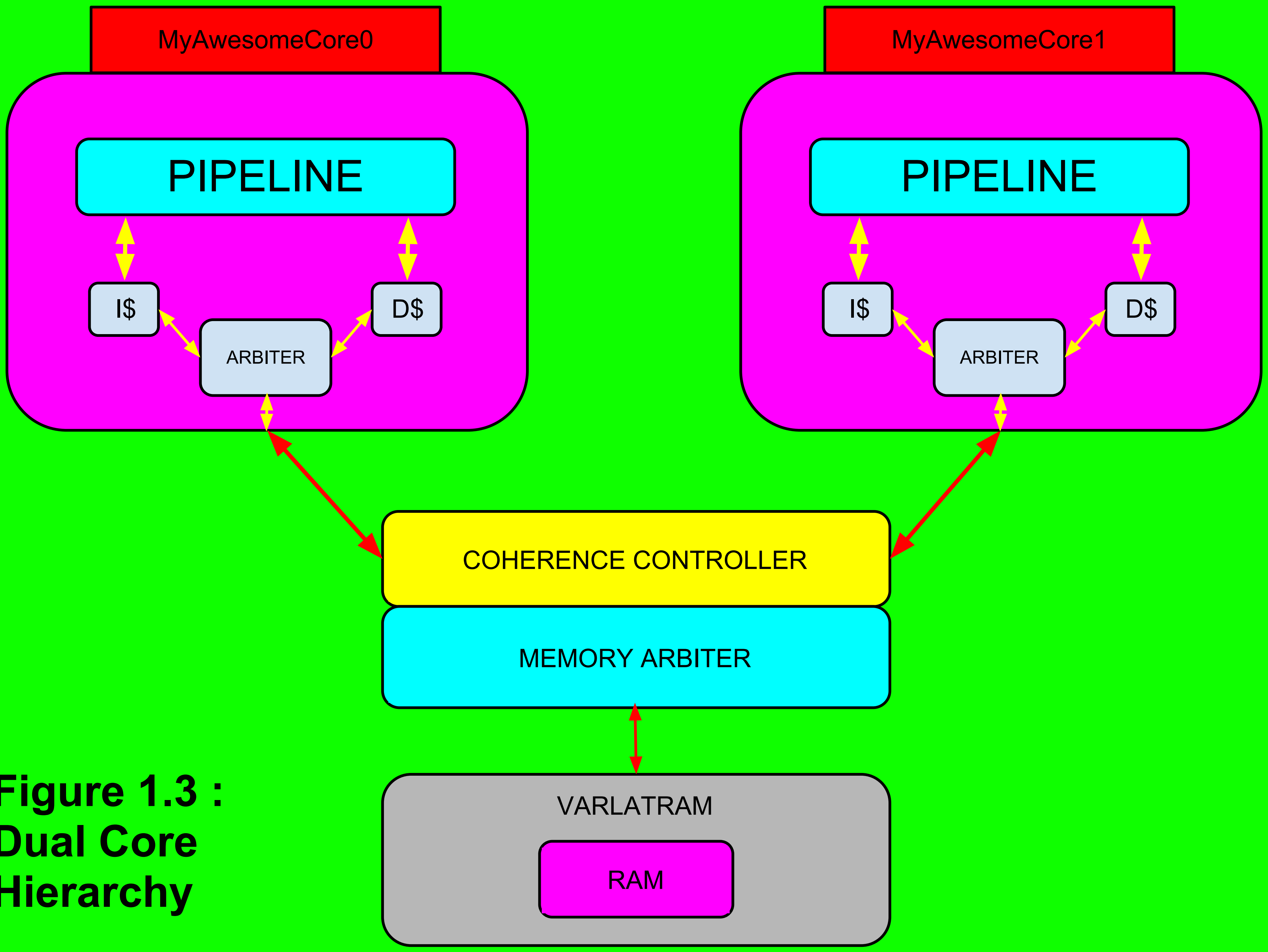


Figure 1.3 : Dual Core Hierarchy

Figure 1.4 - Cache coherence signals

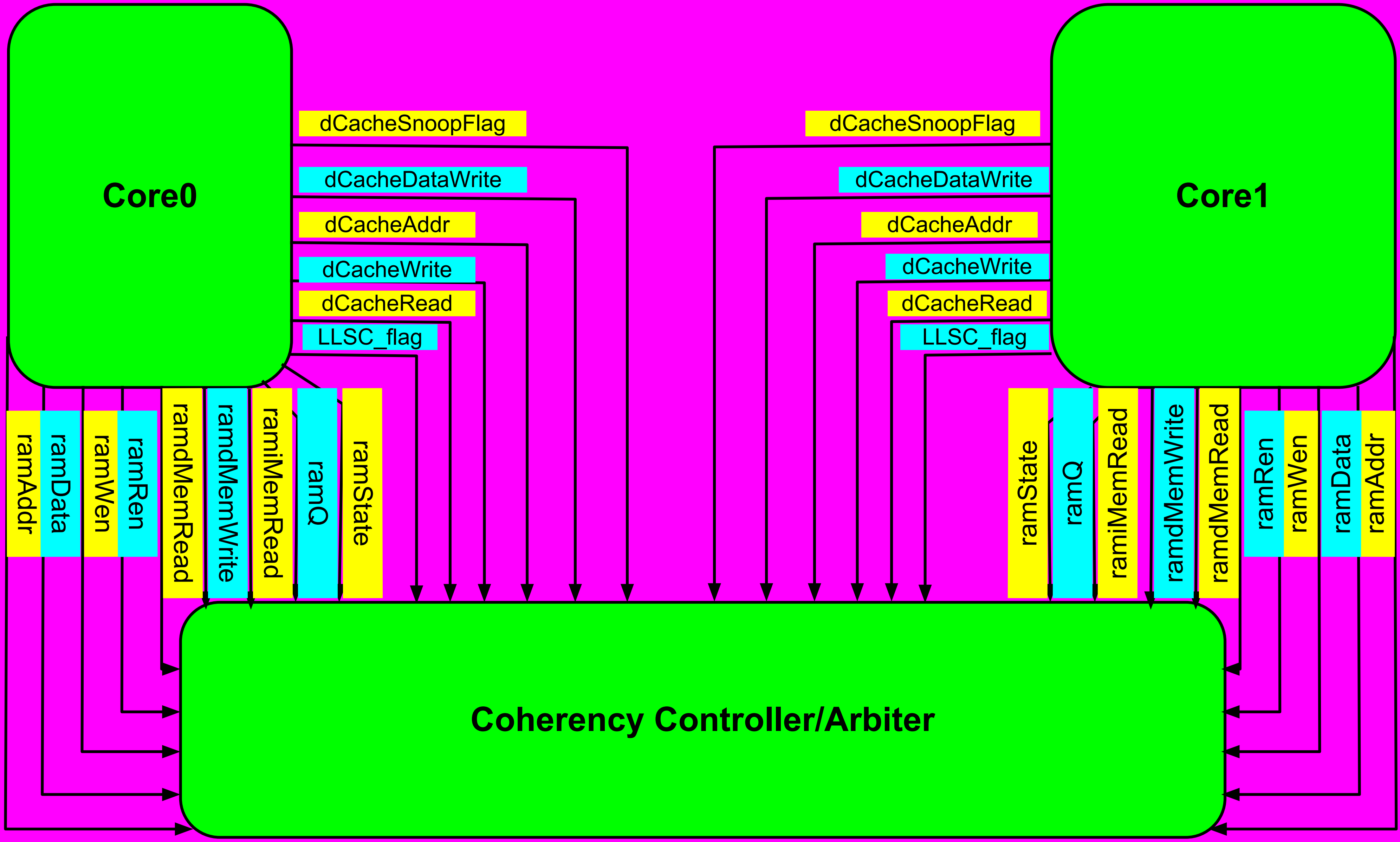
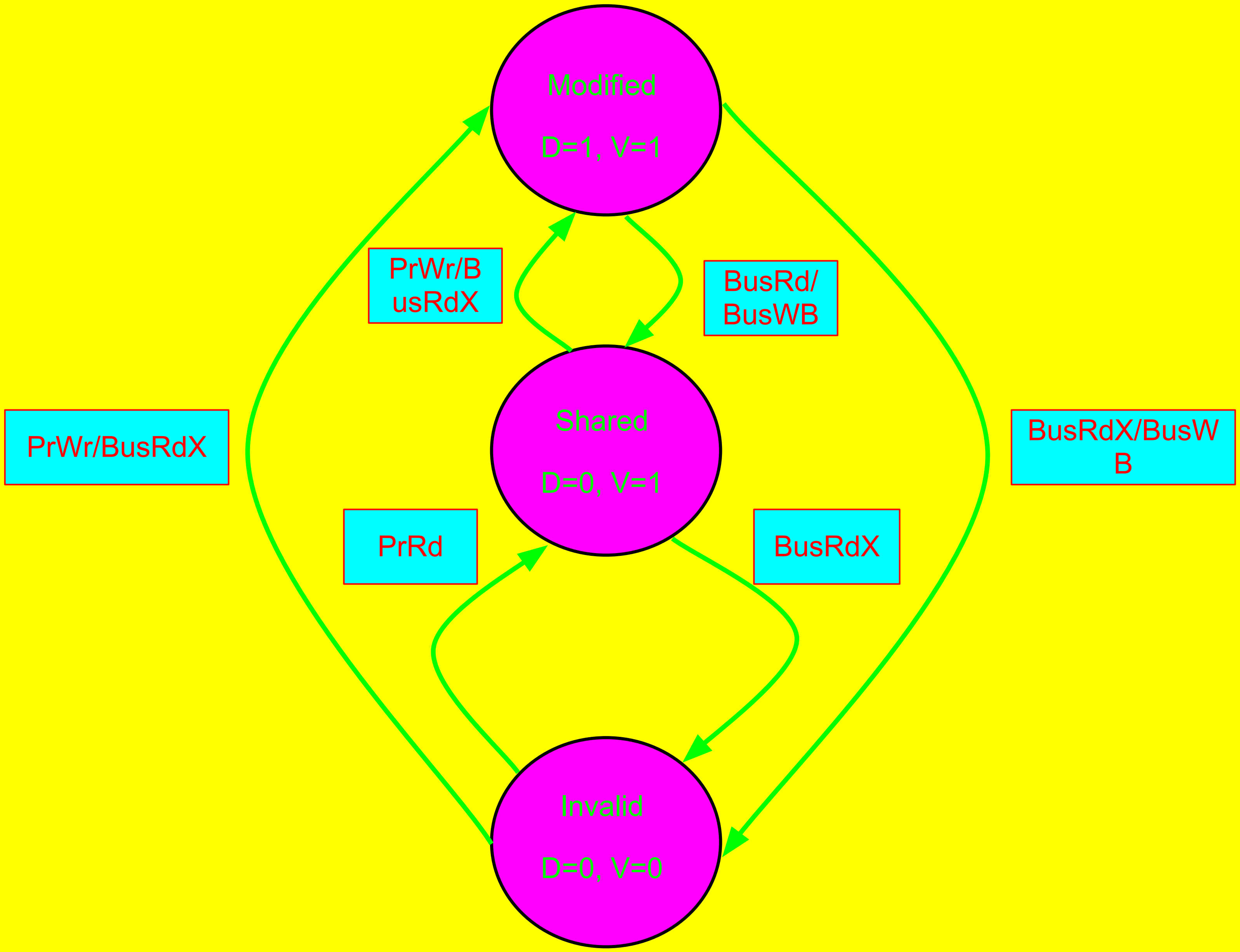


Figure 1.5 - MSI states



Appendix B

1. **Code for dual.mergeSort.asm**

```
      org 0x0000

      ori    $sp, $zero, 0x3ffc      # stack
      ori    $1, $zero, 0xC8 # 49*4bytes numbers to be sorted #int n
      ori    $2, $zero, 0x0004 # for loop counter #int c
      ori    $3, $zero, sortdata # int array[]
      ori    $10, $zero, 0x0004 # used for [d-1]
      jal    outerLoop
waitOnCore1:
      ori    $13, $zero, 0x3124
      lw     $12, 0($13)
      beq    $12, $zero, waitOnCore1
        ori     $12, $zero, 0x0002
        sw      $12, 0($13)

      jal    mergeArrays
      halt

      org 0x0200

      ori    $sp, $zero, 0x3ffc      # stack
      ori    $1, $zero, 0x190 # 100*4bytes numbers to be sorted #int n
      ori    $2, $zero, 0x00CC # for loop counter #int c
      ori    $3, $zero, sortdata # int array[]
```

```
        ori     $10, $zero, 0x0004 # used for [d-1]
        jal     outerLoop
        #ori     $11, $zero, 0x0001
        ori     $12, $zero, 0x3124
        #sw      $11, 0($12)

        ll      $11, 0($12)
        ori     $11, $zero, 0x0001
          sc       $11, 0($12)

        halt

outerLoop:
        beq     $2, $1, endOuterLoop
        addu    $4, $2, $zero # int d


innerLoop:
        beq     $4, $zero, endInnerLoop
getLock:
        lw      $6, sortdata($4) # array[d]
        subu    $5, $4, $10
        lw      $7, sortdata($5) # array[d-1]
        sltu    $8, $6, $7 # array[d] < array[d-1]
        beq     $8, $zero, endInnerLoop
        sw      $6, sortdata($5) # store array[d] to location of array[d-1]
        sw      $7, sortdata($4) # store array[d-1] to location of array[d]
          beq     $7, $zero, getLock

        subu    $4, $4, $10 # d--
        beq     $zero, $zero, innerLoop



endInnerLoop:
        addiu   $2, $2, 0x0004 # increment for loop count by 1
        beq     $zero, $zero, outerLoop

endOuterLoop:
        jr $31

        halt

mergeArrays:
        ori     $1, $zero, sortdata #index0 counter for array sorted on core0
        addiu   $16, $1, 0x00C4 #Upper limit for index0
        addiu   $2, $1, 0x00C8 # index1 counter for array sorted on core1
        addiu   $17, $1, 0x0190 #Upper limit for index1
        ori     $10, $zero, 0x2000 #address to store final merged array to
        addu    $20, $16, $17 # sum of both max indices

mergeLoop:
        addu    $19, $1, $2
        beq     $19, $20, finishedMerging
        lw      $3, 0($1)
        lw      $4, 0($2)
        beq     $1, $16, storeSecondArray
        sltu    $5, $3, $4
        bne     $5, $zero, storeFirstArrayElement
        #otherwise store second ArrayElement to memory
```

```
storeSecondArray:
        sw      $4, 0($10)
        addiu   $10, $10, 0x0004
        beq     $2, $17, mergeLoop
        addiu   $2, $2, 0x0004
        beq     $zero, $zero, mergeLoop

storeFirstArrayElement:
        sw      $3, 0($10)
        addiu   $10, $10, 0x0004
        beq     $1, $16, mergeLoop
        addiu   $1, $1, 0x0004

        beq     $zero, $zero, mergeLoop

finishedMerging:
        jr $31

        org 0x00F00
sortdata:
cfw 0x087d
cfw 0x5fcb
cfw 0xa41a
cfw 0x4109
cfw 0x4522
cfw 0x700f
cfw 0x766d
cfw 0x6f60
cfw 0x8a5e
cfw 0x9580
cfw 0x70a3
cfw 0xaea9
cfw 0x711a
cfw 0x6f81
cfw 0x8f9a
cfw 0x2584
cfw 0xa599
cfw 0x4015
cfw 0xce81
cfw 0xf55b
cfw 0x399e
cfw 0xa23f
cfw 0x3588
cfw 0x33ac
cfw 0xbce7
cfw 0x2a6b
cfw 0x9fa1
cfw 0xc94b
cfw 0xc65b
cfw 0x0068
cfw 0xf499
cfw 0x5f71
cfw 0xd06f
cfw 0x14df
cfw 0x1165
cfw 0xf88d
cfw 0x4ba4
cfw 0x2e74
cfw 0x5c6f
cfw 0xd11e
cfw 0x9222
cfw 0xacdb
cfw 0x1038
cfw 0xab17
cfw 0xf7ce
cfw 0x8a9e
cfw 0x9aa3
cfw 0xb495
cfw 0x8a5e
cfw 0xd859
cfw 0x0bac
cfw 0xd0db
cfw 0x3552
cfw 0xa6b0
cfw 0x727f
cfw 0x28e4
cfw 0xe5cf
cfw 0x163c
cfw 0x3411
cfw 0x8f07
cfw 0xfab7
cfw 0x0f34
cfw 0xdabf
cfw 0x6f6f
cfw 0xc598
cfw 0xf496
cfw 0x9a9a
cfw 0xbd6a
cfw 0x2136
cfw 0x810a
cfw 0xca55
cfw 0x8bce
cfw 0x2ac4
cfw 0xddce
cfw 0xdd06
cfw 0xc4fc
cfw 0xfb2f
cfw 0xee5f
cfw 0xfd30
cfw 0xc540
cfw 0xd5f1
cfw 0xbdad
cfw 0x45c3
cfw 0x708a
cfw 0xa359
cfw 0xf40d
cfw 0xba06
cfw 0xbace
cfw 0xb447
cfw 0x3f48
cfw 0x899e
cfw 0x8084
cfw 0xbdb9
cfw 0xa05a
cfw 0xe225
cfw 0xfb0c
cfw 0xb2b2
cfw 0xa4db
cfw 0x8bf9
cfw 0x12f7
```

2. **Code for example.asm**

```
#-----------------------------------------
# Originally Test and Set example by Eric Villasenor
# Modified to be LL and SC example by Yue Du
#-----------------------------------------

#-------------------------------------------------------
# First Processor
#-------------------------------------------------------
        org         0x0000                              # first
processor p0
        ori         $sp, $zero, 0x3ffc              # stack
        jal         mainp0                              # go to
program
        jal         mainp1                              # go to
program
        jal         mainp0                              # go to
program
        halt

# pass in an address to lock function in argument register 0
# returns when lock is available
lock:
aquire:
        ll          $t0, 0($a0)                     # load lock location
      bne          $t0, $0, aquire                  # wait on lock to be open
      addiu        $t0, $t0, 1
      sc          $t0, 0($a0)
      beq          $t0, $0, lock                    # if sc failed retry
      jr          $ra


# pass in an address to unlock function in argument register 0
# returns when lock is free
unlock:
        sw           $0, 0($a0)
      jr          $ra

# main function does something ugly but demonstrates beautifully
mainp0:
        push          $ra                                 # save return
address
        ori           $a0, $zero, l1                      # move lock to
arguement register
        jal           lock                                # try to aquire the
lock
        # critical code segment
        ori           $t2, $zero, res
        lw            $t0, 0($t2)
        addiu         $t1, $t0, 2
        sw          $t1, 0($t2)
        # critical code segment
        ori         $a0, $zero, l1                      # move lock to
arguement register
        jal         unlock                              # release the lock
        pop         $ra                                 # get return address
        jr          $ra                                 # return to caller
l1:
        cfw    0x0


#--------------------------------------------------------
# Second Processor
#--------------------------------------------------------
        org         0x200                               # second processor
p1
        ori         $sp, $zero, 0x7ffc                  # stack
        jal         mainp1                                      # go to
program
        jal         mainp0                                      # go to
program
        jal         mainp1                                      # go to
program
        halt

# main function does something ugly but demonstrates beautifully
mainp1:
        push        $ra                                 # save return
address
```

```
        ori         $a0, $zero, l1                            # move lock to
arguement register
        jal         lock                                      # try to aquire the
lock
        # critical code segment
        ori         $t2, $zero, res
        lw          $t0, 0($t2)
        addiu       $t1, $t0, 1
        sw          $t1, 0($t2)
        # critical code segment
        ori         $a0, $zero, l1                            # move lock to
arguement register
        jal         unlock                                    # release the lock
        pop         $ra                                       # get return address
        jr          $ra                                       # return to caller

res:
        cfw 0x0                                                   # end result
should be 3
```

3. **Code for dual.mergeSort2.asm**

```
        org 0x0000

        ori $2, $0, 100     # $2 is length of array
        ori $3, $0, 0x1000  # $3 is the head of the array
        ori $4, $0, 50     # $4 is outer loop counter
        ori $5, $0, 0       # $5 is an inner loop counter
        ori $6, $0, 49      # $6 is the outer compare value
        ori $7, $0, 1       # this is a 1
        ori $8, $0, 0x1000  # $8 is the current address being looked at

outer:
        beq $0, $6, done1



        or $5, $0, $6       #store outer loop count into the inner loop counter
        ori $8, $0, 0x1000  # reset address

inner:
        subu $5, $5, $7
        lw $10, 0($8)
        lw $11, 4($8)

        slt $12, $11, $10    #compare the adjacent elements
        beq $12, $0, check1  #if the i+1 element is less than the i element swap
        sw $10, 4($8)
        sw $11, 0($8)

check1:
        addiu $8, $8, 4
        bne $5, $0, inner




        subu $6, $6, $7
```

```
	j outer
done1:

	#do parallelllll-y things

	ori $20, $0, 0x0F00
	ori $9, $0, 1
	ll $14, 0($20)

	#every core adds one to this address and when the first core sees that all cores
	#have added one, it will continue.
addone1:
	ll $14, 0($20)
	beq $14, $0, addone1
	addiu $14, $14, 1
	sc $14, 0($20)
	beq $14, $0, addone1

	#wait for all cores to finish (when the value = 2)
spinwait:
	lw $14, 0($20)
	ori $9, $0, 2
	bne $14, $9, spinwait

#	#threads are now syncronized!!!
#	#DO MERGE SORT NOW
#	ori $2, $0, 0x1000    #first array
#	ori $3, $0, 0x10C8    # second array
#	ori $4, $0, 0x2000    #THE SORTED ARRAY
```

```
#	ori $5, $0, 100		# loop counter
#	ori $6, $0, 0x10C8	#limit of first array
#	ori $8, $0, 0x1190	#limit of the second array
#
#loopguy:
#	lw $10, 0($2)
#	lw $11, 0($3)
#
#	beq $2, $6, storeP2
#	beq $3, $8, storeP1
#
#	slt $12, $11, $10
#	beq $12, $0, storeP1 #if $10 > $11 branch to storeP1 else store P2's val
#storeP2:
#	#store P2
#	sw $11, 0($4)
#	addiu $3, $3, 4	#increment address counter for P2's array
#	j storedone
#
#storeP1:
#	#store P1
#	sw $10, 0($4)
#	addiu $2, $2, 4	#increment address counter for P1's array
#
#
#storedone:
#	addiu $4, $4, 4	# increment the sorted address counter
#	subu $5, $5, $7	#sub 1 from counter
#	bne $5, $0, loopguy
```

```
        halt


#################################SECOND
PROCESS#########################
        org 0x0200

        ori $2, $0, 100     # $2 is length of array
        ori $3, $0, 0x1000  # $3 is the head of the array
        ori $4, $0, 50     # $4 is outer loop counter
        ori $5, $0, 0       # $5 is an inner loop counter
        ori $6, $0, 49      # $6 is the outer compare value
        ori $7, $0, 1       # this is a 1
        ori $8, $0, 0x10C8  # $8 is the current address being looked at

outer2:
        beq $0, $6, done2



        or $5, $0, $6       #store outer loop count into the inner loop counter
        ori $8, $0, 0x10C8  # reset address

inner2:
        subu $5, $5, $7
        lw $10, 0($8)
        lw $11, 4($8)
        slt $12, $11, $10    #compare the adjacent elements
        beq $12, $0, check2  #if the i+1 element is less than the i element swap
        sw $10, 4($8)
        sw $11, 0($8)

check2:
        addiu $8, $8, 4
        bne $5, $0, inner2




        subu $6, $6, $7
        j outer2
done2:


        #do parallelllll-y things PT 2
        ori $20, $0, 0x0F00
        ori $9, $0, 1

        #every core adds one to this address and when the first core sees that all cores
        #have added one, it will continue.
addone2:
        ll $14, 0($20)
        addiu $14, $14, 1
        sc $14, 0($20)
        beq $14, $0, addone2

        halt
        org 0x0F00
link_var:
        cfw 0x0000
#------ DATA VARIABLES----------
        org 0x1000
sortdata:
        cfw 0x087d
        cfw 0x5fcb
        cfw 0xa41a
        cfw 0x4109
        cfw 0x4522
        cfw 0x700f
        cfw 0x766d
        cfw 0x6f60
        cfw 0x8a5e
        cfw 0x9580
        cfw 0x70a3
        cfw 0xaea9
        cfw 0x711a
        cfw 0x6f81
        cfw 0x8f9a
        cfw 0x2584
        cfw 0xa599
        cfw 0x4015
        cfw 0xce81
        cfw 0xf55b
        cfw 0x399e
        cfw 0xa23f
        cfw 0x3588
        cfw 0x33ac
        cfw 0xbce7
        cfw 0x2a6b
        cfw 0x9fa1
        cfw 0xc94b
        cfw 0xc65b
        cfw 0x0068
        cfw 0xf499
        cfw 0x5f71
        cfw 0xd06f
        cfw 0x14df
        cfw 0x1165
        cfw 0xf88d
        cfw 0x4ba4
        cfw 0x2e74
        cfw 0x5c6f
        cfw 0xd11e
        cfw 0x9222
        cfw 0xacdb
        cfw 0x1038
        cfw 0xab17
        cfw 0xf7ce
        cfw 0x8a9e
        cfw 0x9aa3
        cfw 0xb495
        cfw 0x8a5e
        cfw 0xd859
        cfw 0x0bac
        cfw 0xd0db
        cfw 0x3552
        cfw 0xa6b0
        cfw 0x727f
        cfw 0x28e4
        cfw 0xe5cf
        cfw 0x163c
        cfw 0x3411
        cfw 0x8f07
        cfw 0xfab7
        cfw 0x0f34
        cfw 0xdabf
        cfw 0x6f6f
        cfw 0xc598
        cfw 0xf496
        cfw 0x9a9a
        cfw 0xbd6a
        cfw 0x2136
        cfw 0x810a
        cfw 0xca55
        cfw 0x8bce
        cfw 0x2ac4
        cfw 0xddce
        cfw 0xdd06
        cfw 0xc4fc
        cfw 0xfb2f
        cfw 0xee5f
        cfw 0xfd30
        cfw 0xc540
        cfw 0xd5f1
        cfw 0xbdad
        cfw 0x45c3
        cfw 0x708a
        cfw 0xa359
        cfw 0xf40d
        cfw 0xba06
        cfw 0xbace
        cfw 0xb447
        cfw 0x3f48
        cfw 0x899e
        cfw 0x8084
        cfw 0xbdb9
        cfw 0xa05a
        cfw 0xe225
        cfw 0xfb0c
        cfw 0xb2b2
        cfw 0xa4db
        cfw 0x8bf9
        cfw 0x12f7
```