Trevor Morris and Tristan Seroff
5/5/16

ECE 154B Project 2 Report

Introduction
For this project we modified our pipelined MIPS processor to include memory access latency, then designed and integrated a cache unit to improve performance.

Design Methodology
We started with the pipelined MIPS processor we created and tested in Project 1. First, we modified the data memory to include 20 cycles of latency upon access requests. Next, we modified the processor hazard unit to support stalling due to memory latency, and modified the datapath to support this new stall condition. We then tested and debugged the processor using the same test program from Project 1 to ensure correct operation.

Then, we created a 32KB 2-way set associative cache in Verilog code and added it to new a top-level module to contain the memory hierarchy, along with the old data memory module. We substituted the new memory system into the processor in place of the old data memory. We made modifications to a few of the control signals in the processor during debugging to ensure that the new memory system was fully integrated into the pipelined processor. Finally, we tested and compared the performance of the processor design with the single data memory module and the processor design with the cache/data memory hierarchy.

Step 1. Initialize Project
We made a copy of the design files from Project 1 to begin the project.

Step 2. Cache Design
Before making the cache itself, we had to determine how addresses would be formed in order to know the dimensions of the cache. Because we want a byte addressable memory, we need 2 bits in the address for the byte offset. Because each block has 4 words, we need 2 bits in the memory for the block offset. For a 32Kbyte cache with 4-word blocks of 32-bit words, we determined each way would need 2048 blocks. This means the set index in the array must be 11 bits wide (log2(2048) is 11). For a 32-bit address, this leaves 17 bits for the tag. Therefore memory addresses are interpreted in the following format:

31:15	14:4	3:2	1:0
Tag(17 bits)	Index(11 bits)	Block offset(2 bits)	Byte offset(2bits)

To create the cache, we made two modules. One module represents an individual 'way' of the cache, while the higher level module contains all of the ways of the cache and operates on them.

The way module has an internal array of 2048 128-bit blocks which form the data of the cache. Each block also has a valid and a dirty bit, as well as a tag. For reads, the way outputs the valid bit, dirty bit, block, and tag at the supplied index. For writes, the memory location at index and byte offset is replaced with the write data and the dirty bit for this block is set. For loads, the block at the given index is replaced with the load block, the valid bit is set to 1 and the dirty bit is set to 0.

The main cache module contains two copies of the way module for the 2-Way Set Associative Cache. There is an output ‘ready’ flag which goes to the hazard unit and causes a memstall if it is 0. The following logic is performed for reads and writes depending on if there is a hit or miss.

On misses, there are two operations that need to happen: evicts and loads. Old data must be evicted if it is being overwritten by a load. On an evict, the evicted data is copied to a register and evict flag is set. The main memory copies from the register to MM. Loads are when data is copied from MM to cache. On a load, load is set to 1 and the cache begins to wait for a load by setting wait_for_load to 1. On a write miss, wait_for_write is set so that the cache can write to the block after it has been loaded to MM.

Situation	Logic	Ready flag status	Wait_for_load flag status	Wait_for_write flag status
Read hit	Set output to the way that hit	1	0	0
Read miss	If block was dirtry, evict block to MM Load new block from MM	0 until load is done, then 1	1	0
Write hit	Write to way that hit	1	0	0
Write miss	If block was dirtry, evict block to MM Load new block from MM After load, write to new block	0 until load and write is done, then 1	1	1

The upper cache module also has a ‘use’ bit for each set, which corresponds to which way was least recently used. After every hit, the use bit is switched to correspond with the way that did not hit. After every load, the bit is toggled. The use bit is used to select which way to use after any miss.

Step 3. Memory Latency

We first took the (main) data memory module used in Project 1, and modified it by adding reset and operation status signals, and a new internal register to hold a cycle count. In our first design, the cycle count is triggered to begin when the data address on the input port is changed, and read or write operations finish by toggling the status signal after 20 cycles have elapsed.

During later integration, we realized that we needed a more robust signal for the latency counter to start, so we added a discrete control signal to start read/write operations with the data memory module. Additionally, we realized that we could improve performance of the module by adding a second set of address and data ports for write operations, which allows a read and write operation to occur simultaneously, if both control signals are enabled.

Step 4. Data Hazard

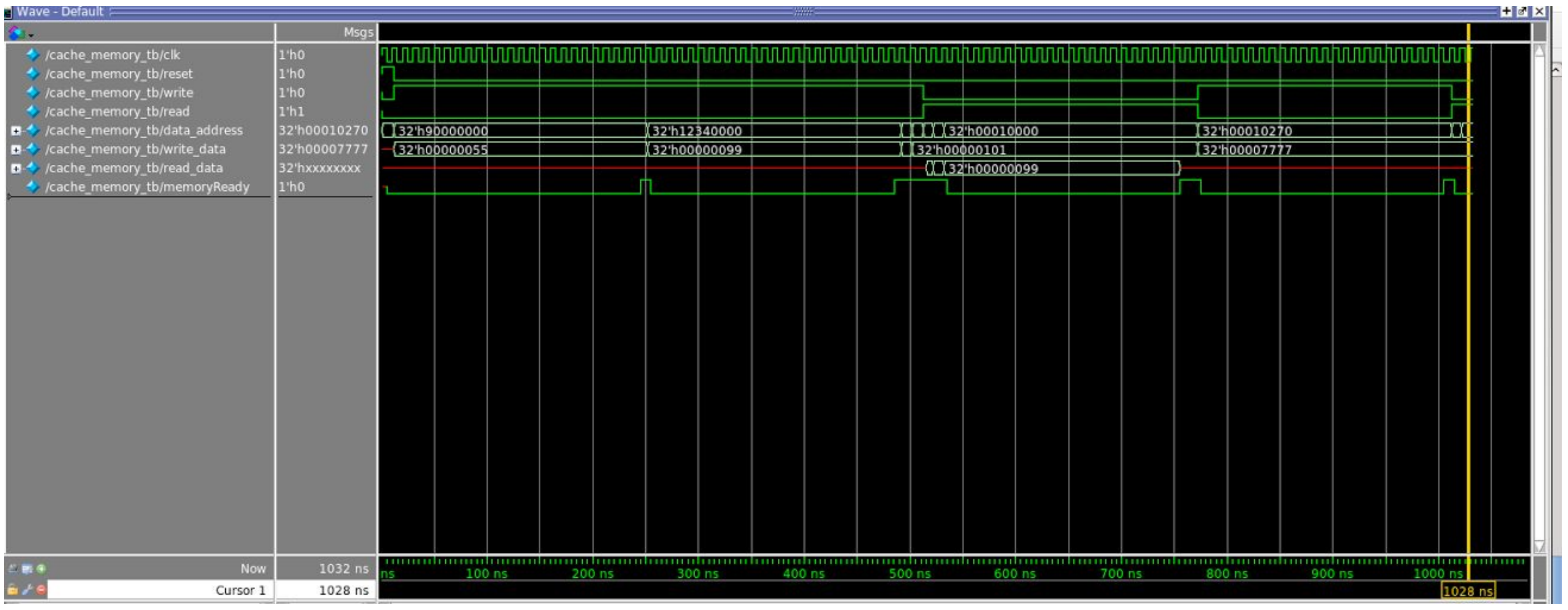
The processor datapath was modified to support the new latency of the main memory by stalling until the memory status signal becomes enabled after the 20-cycle period has elapsed. A new stalling condition, ‘memstall’, was added to the hazard control unit, and the memory status signal was connected to the hazard unit to trigger this condition. The datapath was modified to stall the first four pipeline stages during this stall condition, and to flush the data in the writeback stage repeatedly until the stalling period ended. This required adding enable control ports to the pipeline buffers for the Execute and Memory processor stages, and adding a clear control port to the Writeback stage pipeline buffer. These control ports were connected to new signals from the hazard unit, called StallE, StallM, and FlushW respectively, which operate in a manner consistent with the previously existing StallF, StallD, and FlushE signals.

During testing and debugging, we realized that jump instructions were not being properly stalled - When the jump instruction reached the decode stage, it was decoded and caused the Decode stage buffer to flush its instruction, but when the hazard unit was attempting to perform a stall operation, the instruction being flushed was the jump instruction itself! We solved this issue by re-routing the jump/branch signal, which triggers the Decode stage pipeline buffer to be flushed, through the hazard unit. We modified the hazard unit again to prevent the signal from propagating until the stalling operation is finished.

Step 5. Integrate the Cache

The cache and the main memory module were added to a new top-level data memory module and connected together, and this new top-level module was then connected to the interface of the rest of the processor.

We created a testbench to test the memory system as a whole independent for the rest of the processor. We first do a store and observe a 20+ cycle delay(there is some cache overhead) as the block is initially loaded into the cache. We then do a store to the same block but with a different tag, which evicts the first store and loads the new block. We then test a few writes on words in the new block that was loaded and observe hits. We then read from the address that was evicted earlier and observe a delay and observe the original written value was preserved. We then test that writing to a different block does not affect other blocks.



/cache_memory_tb/clk	1'h0
/cache_memory_tb/reset	1'h0
/cache_memory_tb/write	1'h0
/cache_memory_tb/read	1'h1
/cache_memory_tb/data_address	32'h00010270
/cache_memory_tb/write_data	32'h00007777
/cache_memory_tb/read_data	32'hxxxxxxxx
/cache_memory_tb/memoryReady	1'h0
Now	1032 ns
Cursor 1	1028 ns

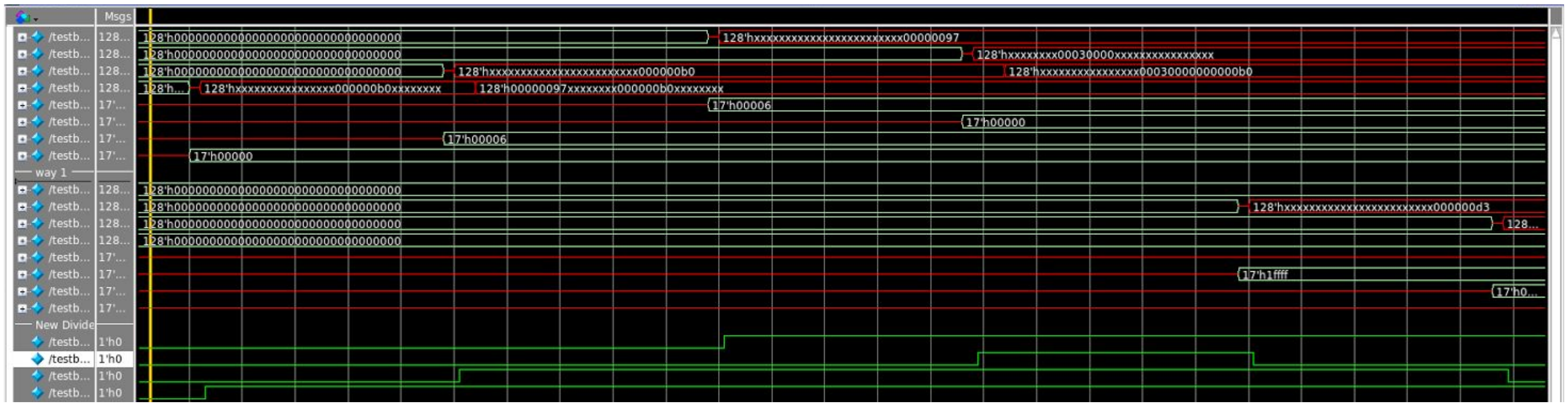
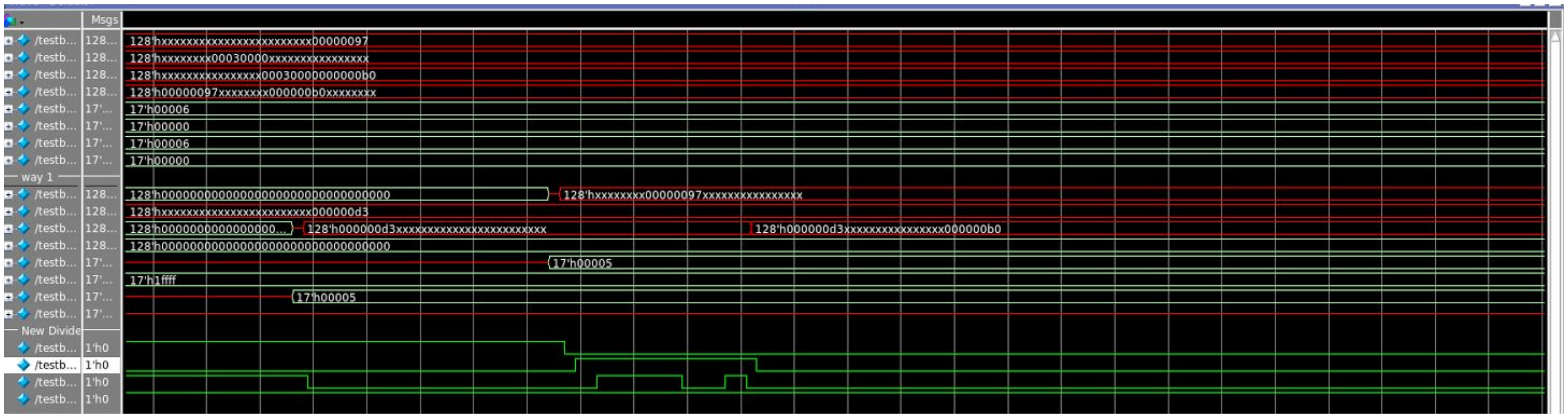
Step 6. Measure Performance

We ran the sample program provided both with the cache and without.

Cache?	Miss Rate	Execution Time
With	33%	2410 ns
Without	N/A	5070 ns

The miss rate of a two-way set associative cache is usually better than that of a direct mapped cache of the same capacity and block size. Average programs will yield a lower miss rate in the two-way set associative cache, but it is possible for certain odd memory access patterns to cause a higher miss rate. One example would be cyclical accesses to three different memory addresses which all map to the same cache block index, which would cause a three-way "ping-pong" effect where the block for each address is evicted the cycle before it needs to be accessed again. The actual performance varies based on the specific memory addresses and the sequence of accesses to them.

The waveforms below show the affected blocks and their tags in each way during the sample program. From the top the blocks are: Block 0x407, 0x403, 0x007, 0x003.


Msgs
/testb...
way 1
New Divide
128'h00000000000000000000000000000000
128'hxxxxxxxxxxxxxxxxxxxxxxxx00000097
128'hxxxxxxxx00030000xxxxxxxxxxxxxxxx
128'hxxxxxxxxxxxxxxxxxxxxxxxx000000b0
128'hxxxxxxxxxxxxxxxx00030000000000b0
128'hxxxxxxxxxxxxxxxx000000b0xxxxxxxx
128'h00000097xxxxxxxx000000b0xxxxxxxx
17'h00006
17'h00000
128'hxxxxxxxxxxxxxxxxxxxxxxxx000000d3
17'h1ffff
1'h0

Msgs
/testb...
way 1
New Divide
128'hxxxxxxxxxxxxxxxxxxxxxxxx00000097
128'hxxxxxxxx00030000xxxxxxxxxxxxxxxx
128'hxxxxxxxxxxxxxxxx00030000000000b0
128'h00000097xxxxxxxx000000b0xxxxxxxx
17'h00006
17'h00000
128'h00000000000000000000000000000000
128'hxxxxxxxx00000097xxxxxxxxxxxxxxxx
128'hxxxxxxxxxxxxxxxxxxxxxxxx000000d3
128'h000000d3xxxxxxxxxxxxxxxxxxxxxxxx
128'h000000d3xxxxxxxxxxxxxxxx000000b0
17'h00005
17'h1ffff
1'h0

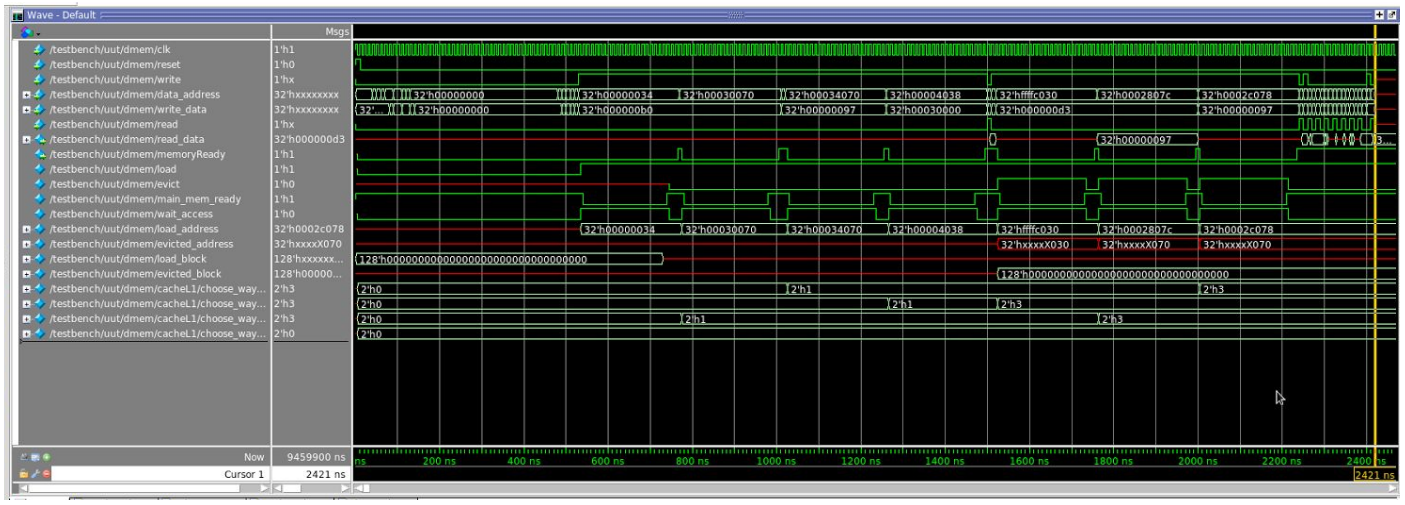
Conclusion

This lab took roughly 15 hours of work. The first large problem we encountered was jump instructions causing themselves to be flushed during stall operations, as described in section “Step 4”. Our cache design went well but integration proved difficult due to timing issues where the cache would signal that the read/write operation was complete but data was still being evicted to/loaded from the main memory module. This issue was eventually solved in debugging through the creation of addition internal control registers in the cache module, as well as a refactoring of the cache module operation logic. Additional issues encountered were typical small Verilog issues such as misspelled connection names or mismatching port sizes.

The final issue we encountered while running the provided test program was several instances where blocks which were evicted from the cache were not being saved in the main memory contents. We examined the addresses of the lost data and discovered that the load-word and store-word instructions which wrote that data all had a 1-bit in the most significant position of their immediate values. Because of the MSB being 1, the immediate was sign-extended to a large negative number which then corresponds to a data address near the top of the 32-bit address space. One example of an instruction that caused this was `sw $t4 0xC030($0)`, which results in the data being saved to the main memory at address 0xFFFFC030. We realized that our main memory module was failing to store data at these high addresses because it was only 64K in size, so we increased the size to 4GB cover the full 32-bit address space. Simulation then caused an overflow error on the 32-bit edition of ModelSim so we switched to the 64-bit edition on the Linux CSIL machines and the problem was resolved.

Extra Credit

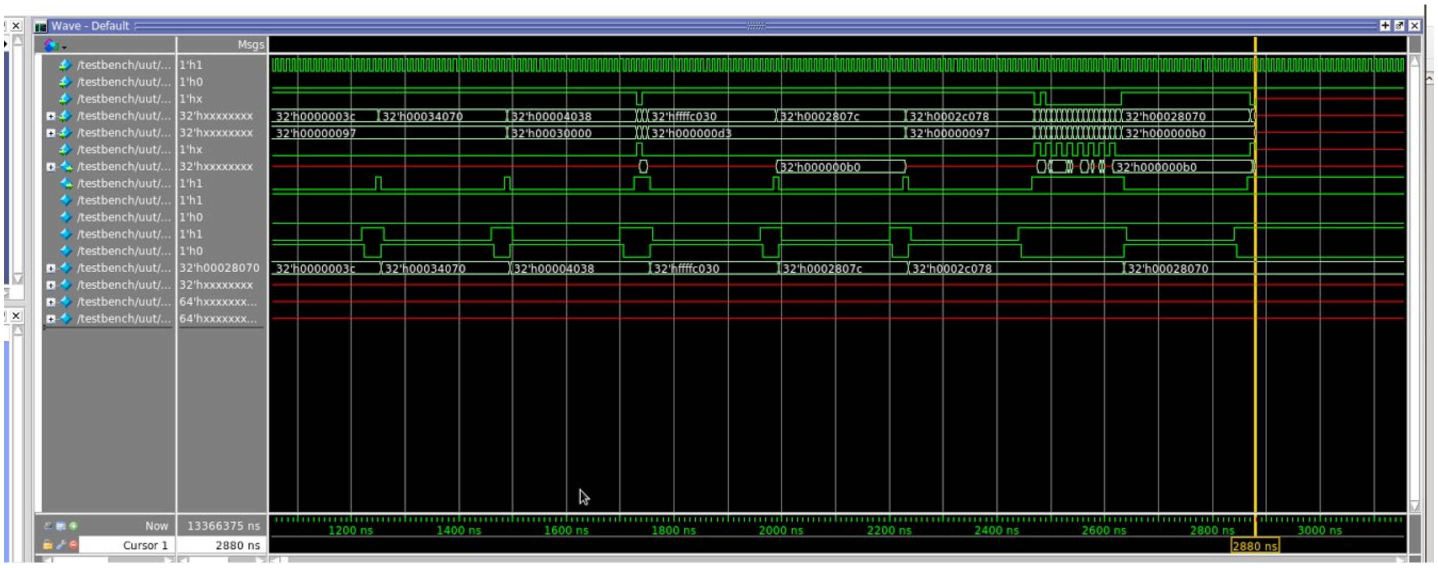
1. Made cache 4-way set associative.
Waveform of 4-way set associative cache running test program.



```
Wave - Default
                                               Msgs
/testbench/uut/dmem/clk                        1'h1
/testbench/uut/dmem/reset                      1'h0
/testbench/uut/dmem/write                      1'hx
/testbench/uut/dmem/data_address               32'hxxxxxxxx
/testbench/uut/dmem/write_data                 32'hxxxxxxxx
/testbench/uut/dmem/read                       1'hx
/testbench/uut/dmem/read_data                  32'h000000d3
/testbench/uut/dmem/memoryReady                1'h1
/testbench/uut/dmem/load                       1'h1
/testbench/uut/dmem/evict                      1'h0
/testbench/uut/dmem/main_mem_ready             1'h1
/testbench/uut/dmem/wait_access                1'h0
/testbench/uut/dmem/load_address               32'h0002c078
/testbench/uut/dmem/evicted_address            32'hxxxxX070
/testbench/uut/dmem/load_block                 128'hxxxxxx...
/testbench/uut/dmem/evicted_block              128'h00000...
/testbench/uut/dmem/cacheL1/choose_way...      2'h3
/testbench/uut/dmem/cacheL1/choose_way...      2'h3
/testbench/uut/dmem/cacheL1/choose_way...      2'h3
/testbench/uut/dmem/cacheL1/choose_way...      2'h0
Now       9459900 ns
Cursor 1  2421 ns
```

2. Block size changed from 128 bits to 64 bits. The miss rate increased due to the cache less efficiently utilizing spatial locality with the smaller block size.

Waveform of 2-word-block cache running test program



```
Wave - Default
                          Msgs
/testbench/uut/...        1'h1
/testbench/uut/...        1'h0
/testbench/uut/...        1'hx
/testbench/uut/...        32'hxxxxxxxx
/testbench/uut/...        32'hxxxxxxxx
/testbench/uut/...        1'hx
/testbench/uut/...        32'hxxxxxxxx
/testbench/uut/...        1'h1
/testbench/uut/...        1'h1
/testbench/uut/...        1'h0
/testbench/uut/...        1'h1
/testbench/uut/...        1'h0
/testbench/uut/...        32'h00028070
/testbench/uut/...        32'hxxxxxxxx
/testbench/uut/...        64'hxxxxxxx...
/testbench/uut/...        64'hxxxxxxx...
Now       13366375 ns
Cursor 1  2880 ns
```

Cache	Miss Rate	Execution time
2-Way Associative 4 Word blocks	33%	2410ns
4-Way Associative 4 Word blocks	38%	2640ns
2-Way Associative 2 Word blocks	42%	2880ns