**Homework #8**

Suppose a memory system is byte-addressable (each address refers to an 8-bit piece of information) and that each memory address is a 32-bit number. The entire memory address can be given the name ***Addr(31:0)***.

The **block offset** is the position within a memory block of a particular byte. Blocks follow an alignment restriction the same as words and half-words do: the first byte of any sized quantity must be at an address divisible by the size of the quantity. So, 4-byte word must start at an address divisible by 4, 2-byte half words must start at an address divisible by 2, and n-byte blocks must start at an address divisible by n. The position of a particular byte within the quantity is then given by the low-order bits of the memory address: ***Addr(1:0)*** gives the four possible byte positions within a word (00, 01, 10, or 11) and ***Addr(0:0)*** gives the two possible byte positions within a half word (0 or 1). In general ***Addr(k-1:0)*** gives the position within a quantity of size n, where n = 2k.

The **block address** is the sequential numbering of each possible block within memory. It is given by the high-order bits of the address ***Addr(31:k)***. So, block 0 always starts at memory address 0x0000 0000. If the block size is 16 bytes such that k = 3, then block 1 would start at memory address 0x0000 0010.

For the following cache block sizes specify the number of bits and the range of bits in the block offset and in the block address. The example of a 16 byte block is given for you.

Block Address Block Offset

Block Size Number of Bits Bit Range Number of Bits Bit Range

16 bytes 28 ***Addr(31:4)*** 4 ***Addr(3:0)***

32 bytes 27 *Addr(31:5)* 5 *Addr(4:0)*

128 bytes 25 *Addr(31:7)* 7 *Addr(6:0)*

8 bytes 29 *Addr(31:3)* 3 *Addr(2:0)*

The block address is in turn divided into a tag field and an index field for each cache in the memory system. The index field specifies which set within the cache might hold the information we are looking for. The tag field of the address we are looking for is compared with the tag field of each of the cache blocks within the set to determine if one of those cache blocks does contain the information we are looking for (the operation of the cache is designed such that there will never be more than one match). The number of bits in the index field is determined by the number of sets in the cache. If there are q sets then we need m index bits such that q = 2m. The number of sets is in turn determined by the total size of the cache in bytes, the number of bytes per cache block, and the associativity of the cache. Dividing the total cache size by the number of bytes per cache block gives the number of blocks in the cache. The associativity tells how many blocks are in each set.

For the following cache sizes, block sizes, and associativities state the number of bits in the tag and index fields along with the bit ranges within the memory address of each. The example of a 16KB 2-way associative cache with 16 bit blocks is given for you.

Tag Field Index Field

Cache Type Number of Bits Bit Range Number of Bits Bit Range

16KB 19 ***Addr(31:13)*** 9 ***Addr(12:4)***

2-way

16 byte block

64KB 16 *Addr(31:16)* 13 *Addr(15:3)*

Direct mapped

8 byte block

8KB 22 *Addr(31:10)* 0 *N/A*

Fully associative

1024 byte block

128KB 17 *Addr(31:15)* 10 *Addr(14:5)*

4-way

32 byte block

Suppose that we have a 32-byte 2-way associative cache with 8 byte blocks. The state of the cache is show below. For each of the memory addresses given determine if the cache will hit or if it will miss. If the cache hits, give the value of the data byte at that address. The first example is given for you.

Set Block Valid Tag Data

0 0 Y 0x0000 010 0x00 0x01 0xFF 0xFE 0x03 0x04 0x77 0x66

0 1 Y 0x0000 0FE 0x50 0x51 0x5F 0x5E 0x23 0x24 0x27 0x26

1 2 Y 0x0000 402 0x40 0x44 0x47 0x33 0x35 0x36 0x37 0x39

1 3 N 0xFFFF 123 0x10 0x11 0x1F 0x1E 0x13 0x14 0x17 0x16

Address Hit/Miss Data

0x0000 0FE2 Hit 0x5F

0xFFFF 1238 Miss Invalid

0x0000 0019 Miss N/A

0x0000 402A Hit 0x47

0x0000 0FE7 Hit 0x26