2016

Computer Architecture Lab

LAB 5: SIMULATING A CACHE

JEFFREY HUANG

JEFFREY HUANG | RUID: 159-00-4687

Assignment 1

```
# Jeffrey Huang
# RUID: 159-00-4687
# NETID: jh1127
# Assignment 1

# Calculate the sum of even numbers inside a given array

            .data 0x10000480
ArrayA:     .word 1, 2, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

            .text
            .globl main
main:
    la $t1, ArrayA
    li $t2, 0                       # counter = 0
    li $t3, 12                      # number of elements in ArrayA
    li $t4, 1                       # mask
    loop:
        lw $t5, 0($t1)
        and $t6, $t5 $t4
        beq $t6, $t4, cont
        add $t2, $t2, $t5
    cont:
        addi $t1, $t1, 4
        addi $t3, $t3, -1
        bgt $t3, $0, loop

    li $v0, 10
    syscall
## ASSIGNMENT 1 PROBLEMS:
#       (a)  PC       = 00000000   EPC      = 00000000   Cause    = 00000000   BadVAddr= 00000000
#            Status   = 3000ff10   HI       = 00000000   LO       = 00000000
#                General Registers
#           R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
#           R1  (at) = 00000000  R9  (t1) = 100004b0  R17 (s1) = 00000000  R25 (t9) = 00000000
#           R2  (v0) = 0000000a  R10 (t2) = 000000bc  R18 (s2) = 00000000  R26 (k0) = 00000000
#           R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
#           R4  (a0) = 00000000  R12 (t4) = 00000001  R20 (s4) = 00000000  R28 (gp) = 10008000
#           R5  (a1) = 00000000  R13 (t5) = 00000090  R21 (s5) = 00000000  R29 (sp) = 7fffe428
#           R6  (a2) = 7fffe430  R14 (t6) = 00000000  R22 (s6) = 00000000  R30 (s8) = 00000000
#           R7  (a3) = 00000000  R15 (t7) = 00000000  R23 (s7) = 00000000  R31 (ra) = 00000000

#           FIR    = 00009800    FCSR    = 00000000    FCCR   = 00000000   FEXR    = 00000000
#           FENR   = 00000000
#                 Double Floating Point Registers
#           FP0  = 0.000000      FP8  = 0.000000      FP16 = 0.000000      FP24 = 0.000000
#           FP2  = 0.000000      FP10 = 0.000000      FP18 = 0.000000      FP26 = 0.000000
#           FP4  = 0.000000      FP12 = 0.000000      FP20 = 0.000000      FP28 = 0.000000
#           FP6  = 0.000000      FP14 = 0.000000      FP22 = 0.000000      FP30 = 0.000000
#                 Single Floating Point Registers
#           FP0  = 0.000000      FP8  = 0.000000      FP16 = 0.000000      FP24 = 0.000000
#           FP1  = 0.000000      FP9  = 0.000000      FP17 = 0.000000      FP25 = 0.000000
#           FP2  = 0.000000      FP10 = 0.000000      FP18 = 0.000000      FP26 = 0.000000
#           FP3  = 0.000000      FP11 = 0.000000      FP19 = 0.000000      FP27 = 0.000000
#           FP4  = 0.000000      FP12 = 0.000000      FP20 = 0.000000      FP28 = 0.000000
#           FP5  = 0.000000      FP13 = 0.000000      FP21 = 0.000000      FP29 = 0.000000
#           FP6  = 0.000000      FP14 = 0.000000      FP22 = 0.000000      FP30 = 0.000000
#           FP7  = 0.000000      FP15 = 0.000000      FP23 = 0.000000      FP31 = 0.000000

#       (b) Set     V   LRU     Tag(h)  Data (h) Way 0                          Acc
#           0       1   0       400012  00000001 00000001 00000002 00000003
#           1       1   0       400012  00000005 00000008 0000000d 00000015
#           2       1   0       400012  00000022 00000037 00000059 00000090     hit
#           3       0   0       400012
#           EXPLANATION:    The cache maps the elements of Array_A as a hexadecimal value in each slot.
#                           Where each set of the cache is capable of storing up to 126 bits of information.
#                           126 bits of information is equivalent to 4 hexadecimal values of size 8.
#                           The first bit in the data determins if the data is a hit or miss, but where when
#                           the data is being retrieved from memory through a tag that is assigned to cache.
#                           In my opinion, the cache is just a sorter that guides the assembler to the right
#                           memory address. Since we have a 256-bit, and a 4-way set, we have 256 divided by 4
#                           which gives us 64 bits divided by 4 which gives us 16 bits per cache line. So when
#                           the decyption of .data, the last two four bits are the cache address, the next two
#                           least significant bits are the index, the and the first two bits are ignored as well.
#                           Thus, the .data 0x10000480 can be converted to binary which results in:
#                           0001 0000 0000 0000 0000 0100 1000 0000. With the bits removed as mentioned above, we
#                           get 0100 0000 0000 0000 0001 0010. Which will result in the tag of 400012. Once the
#                           Data (h) Way 0 is filled, the tag will increment to 400013 which is a incremented by 4
#                           in binary.
```

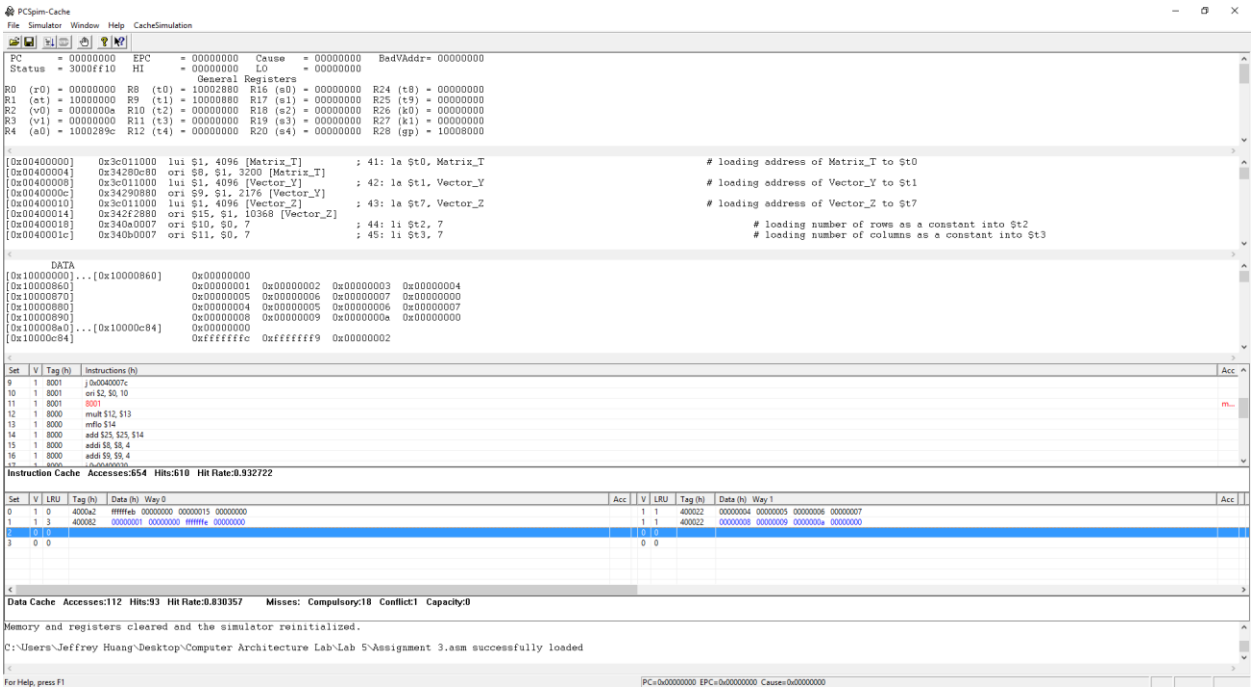
```
#       (b) Set     V   LRU     Tag(h)  Data (h) Way 0                          Acc
#           0       1   0       400012  00000001 00000001 00000002 00000003
#           1       1   0       400012  00000005 00000008 0000000d 00000015
#           2       1   0       400012  00000022 00000037 00000059 00000090     hit
#           3       0   0       400012
#           EXPLANATION:    The cache maps the elements of Array_A as a hexadecimal value in each slot.
#                           Where each set of the cache is capable of storing up to 126 bits of information.
#                           126 bits of information is equivalent to 4 hexadecimal values of size 8.
#                           The first bit in the data determins if the data is a hit or miss, but where when
#                           the data is being retrieved from memory through a tag that is assigned to cache.
#                           In my opinion, the cache is just a sorter that guides the assembler to the right
#                           memory address. Since we have a 256-bit, and a 4-way set, we have 256 divided by 4
#                           which gives us 64 bits divided by 4 which gives us 16 bits per cache line. So when
#                           the decyption of .data, the last two four bits are the cache address, the next two
#                           least significant bits are the index, the and the first two bits are ignored as well.
#                           Thus, the .data 0x10000480 can be converted to binary which results in:
#                           0001 0000 0000 0000 0000 0100 1000 0000. With the bits removed as mentioned above, we
#                           get 0100 0000 0000 0000 0001 0010. Which will result in the tag of 400012. Once the
#                           Data (h) Way 0 is filled, the tag will increment to 400013 which is a incremented by 4
#                           in binary.

#       (c) Set     V   Tag (h)     Instructions (h)
#           0       1   8000        lui $1, 4096
#           1       1   8000        ori $9, $1, 1152
#           2       1   8000        ori $10, $0, 0
#           3       1   8000        ori $11, $0, 12
#           4       1   8000        ori $12, $0, 1
#           5       1   8000        lw $13, 0($9)
#           6       1   8000        and $14, $13,$12
#           7       1   8000        beq $14, $12, 8
#           8       1   8000        add $10, $10, $13
#           9       1   8000        addi $9, $9, 4
#           10      1   8000        addi $11, $11, -1
#           11      1   8000        slt $1, $0, $11
#           12      1   8000        bne $1, $0, -28
#           13      1   8000        ori $2, $0, 10
#           14      1   8000        8000
#           EXPLANATION:    The contents of the instruction cache obtains the instructions through reading
#                           the actual program and converting the user registers to the machine registers.
#                           These values of the instruction cache are acquired from the moment the instruction
#                           is read, determined if the instruction is valid or not. If the instructino is valid,
#                           the cache will miss and then store the instruction into the instruction cache. The
#                           number of access is 96 because of the loops that access the number of times to run
#                           run the program. Then the number of hits is the number of times that the instruction
#                           accessed from the cache.
```

Assignment 2

```
# Jeffrey Huang
# RUID: 159-00-4687
# NETID: jh1127
# Assignment 2

# Registers:        $t0 -> address of ArrayA
#                   $t1 -> address of ArrayB
#                   $t2 -> address of ArrayC
#                   $t3 -> size of ArrayA
#                   $t4 -> size of ArrayB
#                   $t5 -> A[index]
#                   $t6 -> B[index]
#                   $t7 -> C[index]

            .data 0x10000480
ArrayA:     .word       1, 2, 3
ArrayB:     .word       8, 7, 6
ArrayC:     .space      36
space:      .asciiz     " "

            .text
            .globl main
main:
    la $t0, ArrayA                  # loading address of ArrayA into register $t0
    la $t2, ArrayC                  # loading address of ArrayC into register $t2
    li $t3, 3                       # loading array size of ArrayA into register $s0

outerLoop:
    beqz $t3, printSetup            # if $t3 == 0, branch to printSetup
    la $t1, ArrayB                  # loading address of ArrayB into register $t1
    li $t4, 3                       # loading array size of ArrayB into register $s1
    sub $t3, $t3, 1                 # (size of ArrayA)--
    lw $t5, 0($t0)                  # A[index]
    addi $t0, $t0, 4                # incrementing address of ArrayB
innerLoop:
    beqz $t4, outerLoop             # if $t4 == 0, branch to outerLoop
    lw $t6, 0($t1)                  # B[index]
    mult $t6, $t5                   # A[index] * B[index]
    mflo $t7                        # moving product from $LO
    sw $t7, 0($t2)                  # storing product into C[index]
    sub $t4, $t4, 1                 # (size of ArrayB)--
    addi $t1, $t1, 4                # incrementing address of ArrayB
    addi $t2, $t2, 4                # incrementing address of ArrayC
    j innerLoop                     # jump to innerLoop

printSetup:
    la $t2, ArrayC                  # reloading address of ArrayC into register $t2
    li $t3, 9                       # loading size of ArrayC
printLoop:
    beqz $t3, exit                  # if $t3 == 0, branch to exit
    lw $a0, 0($t2)                  # loading syscall argument for print_integer
    li $v0, 1                       # loading syscall service for print_integer
    syscall                         # making syscall to print_integer
    la $a0, space                   # loading syscall argument for print_string
    li $v0, 4                       # loading syscall service for print_string
    syscall
    addi $t2, $t2, 4                # incrementing address of ArrayC
    sub $t3, $t3, 1                 # decrementing size of ArrayC
    j printLoop                     # jump to printLoop
exit:
    li $v0, 10                      # load syscall service for exit_program
    syscall                         # making syscall to exit_program

## ASSIGNMENT 2 PROBLEM:
#           Set     V   LRU     Tag(h)  Data (h) Way 0                          Acc
#           0       1   0       400012  00000001 00000002 00000003 00000008
#           1       1   0       400012  00000007 00000006 00000008 00000007
#           2       1   0       400012  00000006 00000010 0000000e 0000000c
#           3       0   0       400012  00000018 00000015 00000012 00000020     hit
#           EXPLANATION:    The elements of A and B are mapped in each slot in the data cache right next to
#                           to each other. Since the cache isn't filled from one array, the elements of ArrayA
#                           will leave one open slot for the first element of ArrayB. This means that the cache
#                           utilizes each space regardless of whether or not the elements are part of different
#                           arrays.
```

Assignment 3

```
# Jeffrey Huang
# RUID: 159-00-4687
# NETID: jh1127
# Assignment 3


# Registers:        $t0 -> base address of Matrix_T
#                   $t1 -> base address of Vector_Y
#                   $t2 -> number of rows for Matrix_T
#                   $t3 -> number of columns for Matrix_T
#                   $t4 -> Matrix_T($t2, $t3)
#                   $t5 -> Vector_Y($t3))
#                   $t6 ->
#                   $t7 ->

        .data 0x10000860
Vector_X:   .word   1, 2, 3, 4, 5, 6, 7
        .data 0x10000880
Vector_Y:   .word   4, 5, 6, 7, 8, 9, 10
        .data 0x10000c80
Matrix_T:   .word   0, -4, -7, 2, -6, 5, 3
        .data 0x10001080
            .word   4, 0, -5, -1, 3, -7, 6
        .data 0x10001480
            .word   7, 5, 0, -6, -2, 4, -1
        .data 0x10001880
            .word   -2, 1, 6, 0, -7, -3, 5
        .data 0x10001c80
            .word   6, -3, 2, 7, 0, -1, -4
        .data 0x10002080
            .word   -5, 7, -4, 3, 1, 0, -2
        .data 0x10002480
            .word   -3, -6, 1, -5, 4, 2, 0
        .data 0x10002880
Vector_Z:   .word   0, 0, 0, 0, 0, 0, 0
space:      .asciiz " "

        .text 0x00400000
            .globl main
main:
    la $t0, Matrix_T                    # loading address of Matrix_T to $t0
    la $t1, Vector_Y                    # loading address of Vector_Y to $t1
    la $t7, Vector_Z                    # loading address of Vector_Z to $t7
    li $t2, 7                           # loading number of rows as a constant into $t2
    li $t3, 7                           # loading number of columns as a constant into $t3

loop:
    beqz $t3, moveToNextRow             # if $t3 == 0, branch to moveToNextRow
    lw $t4, 0($t0)                      # loading element of Matrix_T
    lw $t5, 0($t1)                      # loading element of Vector_Y
    addi $t3, $t3, -1                   # decrementing number of elements left to multiply
    mult $t4, $t5                       # Vector_Y[x,y] * Matrix_T[x,y]
    mflo $t6                            # getting product from mult function
    add $t9, $t9, $t6                   # adding new product to sum
    addi $t0, $t0, 4                    # incrementing address of Matrix_T
    addi $t1, $t1, 4                    # incrementing address of Vector_Y
    j loop                              # jump to loop
moveToNextRow:
    addi $t2, $t2, -1                   # decrementing number of rows left to multiply
    sw $t9, 0($t7)                      # storing partial result into Vector_Z[x,y]
    li $t9, 0                           # loading initial sum value of each row.
    addi $t7, $t7, 4                    # incrementing to the next empty element
    addi $t0, $t0, 996                  # incrementing to the next row.
    la $t1, Vector_Y                    # reloading the base address to the register
    li $t3, 7                           # reloading the number of elements left to multipy
    beqz $t2, printSetup                # if $t2 == 0, branch to printSetup
    j loop                              # else jump to loop

printSetup:
    la $t7, Vector_Z                    # reloading base address to Vector_Z
    li $t3, 7                           # loading counter to size of Vector_Z

printVector:
    lw $a0, 0($t7)                      # loading element from Vector_Z to $t9
    li $v0, 1                           # loading syscall service for print_int
    syscall                             # making syscall to print_int
    la $a0, space                       # loading blank space for print_string
    li $v0, 4                           # loading syscall service for print_string
    syscall                             # making syscall to print_string
    addi $t3, $t3, -1                   # decrementing the number of elements remaining
    beqz $t3, exit                      # if $t3 == 0, branch to exit
    addi $t7, $t7, 4                    # incrementing to the next element in the vector
    j printVector                       # jump to printVector

exit:
    li $v0, 10                          # loading syscall service for end_program
    syscall                             # making syscall service to end_program
```




```
        DATA
[0x10000000]...[0x10000860]     0x00000000
[0x10000860]                    0x00000001  0x00000002  0x00000003  0x00000004
[0x10000870]                    0x00000005  0x00000006  0x00000007  0x00000000
[0x10000880]                    0x00000004  0x00000005  0x00000006  0x00000007
[0x10000890]                    0x00000008  0x00000009  0x0000000a  0x00000000
[0x100008a0]...[0x10000c84]     0x00000000
[0x10000c84]                    0xfffffffc  0xfffffff9  0x00000002
[0x10000c90]                    0xfffffffa  0x00000005  0x00000003  0x00000000
[0x10000ca0]...[0x10001080]     0x00000000
[0x10001080]                    0x00000004  0x00000000  0xfffffffb  0xffffffff
[0x10001090]                    0x00000003  0xfffffff9  0x00000006  0x00000000
[0x100010a0]...[0x10001480]     0x00000000
[0x10001480]                    0x00000007  0x00000005  0x00000000  0xfffffffa
[0x10001490]                    0xfffffffe  0x00000004  0xffffffff  0x00000000
[0x100014a0]...[0x10001880]     0x00000000
[0x10001880]                    0xfffffffe  0x00000001  0x00000006  0x00000000
[0x10001890]                    0xfffffff9  0xfffffffd  0x00000005  0x00000000
[0x100018a0]...[0x10001c80]     0x00000000
[0x10001c80]                    0x00000006  0xfffffffd  0x00000002  0x00000007
[0x10001c90]                    0x00000000  0xffffffff  0xfffffffc  0x00000000
[0x10001ca0]...[0x10002080]     0x00000000
[0x10002080]                    0xfffffffb  0x00000007  0xfffffffc  0x00000003
[0x10002090]                    0x00000001  0x00000000  0xfffffffe  0x00000000
[0x100020a0]...[0x10002480]     0x00000000
[0x10002480]                    0xfffffffd  0xfffffffa  0x00000001  0xfffffffb
[0x10002490]                    0x00000004  0x00000002  0x00000000  0x00000000
[0x100024a0]...[0x1000289c]     0x00000000
[0x1000289c]                    0x00000020
[0x100028a0]...[0x10040000]     0x00000000
```

Assignment 4

```
# Jeffrey Huang
# RUID: 159-00-4687
# NETID: jh1127
# Assignment 4

            .data 0x10000800
OrinRow_0:  .word    1,  2,  3,  4,  5,  6
OrinRow_1:  .word    7,  8,  9, 10, 11, 12
OrinRow_2:  .word   13, 14, 15, 16, 17, 18
OrinRow_3:  .word   19, 20, 21, 22, 23, 24
OrinRow_4:  .word   25, 26, 27, 28, 29, 30
OrinRow_5:  .word   31, 32, 33, 34, 35, 36

            .data 0x10000900
TransRow_0: .word    0,  0,  0,  0,  0,  0
TransRow_1: .word    0,  0,  0,  0,  0,  0
TransRow_2: .word    0,  0,  0,  0,  0,  0
TransRow_3: .word    0,  0,  0,  0,  0,  0
TransRow_4: .word    0,  0,  0,  0,  0,  0
TransRow_5: .word    0,  0,  0,  0,  0,  0

            .text 0x00400000
main:
    la $t0, OrinRow_0               # loading first row to $t0
    la $s0, TransRow_0              # loading first row to $s0
    la $s1, TransRow_1              # loading second row to $s1
    la $s2, TransRow_2              # loading third row to $s2
    la $s3, TransRow_3              # loading fourth row to $s3
    la $s4, TransRow_4              # loading fifth row to $s4
    la $s5, TransRow_5              # loading sixth row to $s5
    li $t6, 7                       # initializing counter to 7
loop:
    lw $t8, 0($t0)                  # loading element from original
    sw $t8, 0($s0)                  # storing element into transposed
    addi $t0, $t0, 4                # incrementing index of original
    lw $t8, 0($t0)                  # loading element from original
    sw $t8, 0($s1)                  # storing element into transposed
    addi $t0, $t0, 4                # incrementing index of original
    lw $t8, 0($t0)                  # loading element from original
    sw $t8, 0($s2)                  # storing element into transposed
    addi $t0, $t0, 4                # incrementing index of original
    lw $t8, 0($t0)                  # loading element from original
    sw $t8, 0($s3)                  # storing element into transposed
    addi $t0, $t0, 4                # incrementing index of original
    lw $t8, 0($t0)                  # loading element from original
    sw $t8, 0($s4)                  # storing element into transposed
    addi $t0, $t0, 4                # incrementing index of original
    lw $t8, 0($t0)                  # loading element from original
    sw $t8, 0($s5)                  # storing element into transposed
    addi $t0, $t0, 4                # incrementing index of original
    addi $t6, $t6, -1               # decrementing counter
    addi $s0, $s0, 4                # incrementing index of transpose
    addi $s1, $s1, 4                # incrementing index of transpose
    addi $s2, $s2, 4                # incrementing index of transpose
    addi $s3, $s3, 4                # incrementing index of transpose
    addi $s4, $s4, 4                # incrementing index of transpose
    addi $s5, $s5, 4                # incrementing index of transpose
    beq $t6, 6, row1                # changing all values to neccessary
    beq $t6, 5, row2                # changing all values to neccessary
    beq $t6, 4, row3                # changing all values to neccessary
    beq $t6, 3, row4                # changing all values to neccessary
    beq $t6, 2, row5                # changing all values to neccessary
    beq $t6, 1, exit                # changing all values to neccessary

row1:
    la $t0, OrinRow_1               # loading second row to $t0
    j loop                          # jump to loop

row2:
    la $t0, OrinRow_2               # loading third row to $t0
    j loop                          # jump to loop

row3:
    la $t0, OrinRow_3               # loading fourth row to $t0
    j loop                          # jump to loop

row4:
    la $t0, OrinRow_4               # loading fifth row to $t0
    j loop                          # jump to loop

row5:
    la $t0, OrinRow_5               # loading sixth row to $t0
    j loop                          # jump to loop

exit:
    li $v0, 10                      # load syscall service for end_program
    syscall                         # making syscall to end_program
```

Assignment 5

Cache Size: 128B, Block size: 4B, direct-mapping

Set	V	Tag (h)	Instructions (h)	Acc
11	1	8001	beq $1, $14, 48	
12	1	8001	ori $1, $0, 4	
13	1	8001	beq $1, $14, 52	
14	1	8001	ori $1, $0, 3	
15	1	8001	beq $1, $14, 56	
16	1	8001	ori $1, $0, 2	
17	1	8001	beq $1, $14, 60	
18	1	8001	ori $1, $0, 1	

Instruction Cache Accesses:224 Hits:132 Hit Rate:0.589286

Set	V	Tag (h)	Data (h)	Acc
1	1	200011	00000022	
2	1	200011	00000023	
3	1	200013	00000024	m...
4	1	200012	00000019	
5	1	200012	0000001f	
6	1	200010	00000007	
7	1	200012	00000008	
8	1	200012	0000000e	

Data Cache Accesses:72 Hits:0 Hit Rate:0.000000 Misses: Compulsory:72 Conflict:0 Capacity:0

Cache Size: 128B, Block size: 8B, direct-mapping

Set	V	Tag (h)	Instructions (h)	Acc
2	1	8001	addi $16, $16, 4	
3	1	8002	ori $2, $0, 10	
4	1	8002	8002	m...
5	1	8001	addi $19, $19, 4	
6	1	8001	addi $20, $20, 4	
7	1	8001	addi $21, $21, 4	
8	1	8001	ori $1, $0, 6	
9	1	8001	beql $1, $14, 44	

Instruction Cache Accesses:224 Hits:132 Hit Rate:0.589286

Set	V	Tag (h)	Data (h)	Acc
0	1	200011	00000021 00000022	
1	1	200013	0000001e 00000024	m...
2	1	200012	00000019 0000001f	
3	1	200012	00000002 00000008	
4	1	200012	0000000e 00000014	
5	1	200012	0000001a 00000020	
6	1	200010	0000000d 0000000e	
7	1	200012	0000000f 00000015	

Data Cache Accesses:72 Hits:27 Hit Rate:0.375000 Misses: Compulsory:36 Conflict:6 Capacity:3

Cache Size: 128B, Block size: 16B, direct-mapping

Set	V	Tag (h)	Instructions (h)	Acc
2	1	8001	addi $16, $16, 4	
3	1	8002	ori $2, $0, 10	
4	1	8002	8002	m...
5	1	8001	addi $19, $19, 4	
6	1	8001	addi $20, $20, 4	
7	1	8001	addi $21, $21, 4	
8	1	8001	ori $1, $0, 6	
9	1	8001	beq $1, $14, 44	

Instruction Cache Accesses:224 Hits:132 Hit Rate:0.589286

Set	V	Tag (h)	Data (h)	Acc
0	1	200013	00000012 00000018 0000001e 00000024	m...
1	1	200012	00000019 0000001f 00000002 00000008	
2	1	200012	0000000e 00000014 0000001a 00000020	
3	1	200012	00000003 00000009 0000000f 00000015	
4	1	200012	0000001b 00000021 00000004 0000000a	
5	1	200012	00000010 00000016 0000001c 00000022	
6	1	200010	00000019 0000001a 0000001b 0000001c	
7	1	200012	0000001d 00000023 00000006 0000000c	

Data Cache Accesses:72 Hits:38 Hit Rate:0.527778 Misses: Compulsory:18 Conflict:5 Capacity:11

The higher the number of misses half every time the block size is increased. Hence, that means that there are fewer changes in the larger block. This means that the hit rate increases and result in the smaller amount of accessing the memory. Since the cache size remains a constant, the block size allows more information to be stored in one block. As a result, the misses would decrease because the information stored in one block will go up by a power of two. This also means that the hit rate will increase every time the block size increases.