4.31.1.

Instruction \ Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
nop																											
li x12, 0	IF	ID	EX	MEM	WB																						
nop																											
jal ENT		IF	ID	EX	MEM	WB																					
nop																											
ENT: bne x12, x13, TOP			IF	ID	EX	MEM	WB																				
nop																											
TOP: slli x5, x12, 3				IF	ID	EX	MEM	WB																			
nop																											
add x6, x10, x5					IF	ID	EX	MEM	WB																		
ld x7, 0(x6)						IF	ID	EX	MEM	WB																	
nop																											
ld x29, 8(x6)							IF	ID	EX	MEM	WB																
nop																											
nop																											
sub x30, x7, x29								IF	ID	O	EX	MEM	WB														
nop																											
add x31, x11, x5									IF	O	ID	EX	MEM	WB													
sd x30, 0(x31)											IF	ID	EX	MEM	WB												
addi x12, x12, 2											IF	ID	EX	MEM	WB												
nop																											
ENT: bne x12, x13, TOP												IF	ID	O	EX	MEM	WB										
nop																											
TOP: slli x5, x12, 3													IF	O	ID	EX	MEM	WB									
nop																											
add x6, x10, x5															IF	ID	EX	MEM	WB								
ld x7, 0(x6)																IF	ID	EX	MEM	WB							
nop																											
ld x29, 8(x6)																	IF	ID	EX	MEM	WB						
nop																											
nop																											
sub x30, x7, x29																		IF	ID	O	EX	MEM	WB				
nop																											
add x31, x11, x5																			IF	O	ID	EX	MEM	WB			
sd x30, 0(x31)																					IF	ID	EX	MEM	WB		
addi x12, x12, 2																					IF	ID	EX	MEM	WB		
nop																											
ENT: bne x12, x13, TOP																						IF	ID	O	EX	MEM	WB
finish																											

Because in the packet, one instruction must be a memory operation and the other must be an arithmetic/logic / branch instruction, I insert nop if no appropriate instruction can form a packet.

"O" in the diagram means bubble because of hazard.

4.31.2. A two-issue processor has no speedup than a one-issue processor in this case.
The bubbles generated during execution offset the little speedup brought by two-issue processor.

4.31.3

```
      beqz   x13, DONE
      li     x12, 0
TOP:  slli   x5, x12, 3
      add    x6, x10, x5
      ld     x7, 0(x6)
      ld     x29, 8(x6)
      add    x31, x11, x5
      sub    x30, x7, x29
      addi   x12, x12, 2
      sd     x30, 0(x31)
      bne    x12, x13, TOP
DONE:
```

4.31.4.

```
      beqz   x13, DONE
      li     x12, 0
TOP:  slli   x5, x12, 3
      add    x6, x10, x5
      ld     x7, 0(x6)
      add    x31, x11, x5
      ld     x29, 8(x6)
      addi   x12, x12, 2
      sub    x30, x7, x29
      sd     x30, 0(x31)
      bne    x12, x13, TOP
DONE:
```

instructions in the middle of two dashed lines are in the same packet.

4.31.5.

Instruction \ Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
nop																				
beqz x13, NONE	IF	ID	EX	MEM	WB															
nop																				
li x12, 0		IF	ID	EX	MEM	WB														
nop																				
TOP: slli x5, x12, 3			IF	ID	EX	MEM	WB													
nop																				
add x6, x10, x5				IF	ID	EX	MEM	WB												
ld x7, 0(x6)					IF	ID	EX	MEM	WB											
add x31, x11, x5					IF	ID	EX	MEM	WB											
ld x29, 8(x6)						IF	ID	EX	MEM	WB										
addi x12, x12, 2						IF	ID	EX	MEM	WB										
nop																				
sub x30, x7, x29							IF	ID	○	EX	MEM	WB								
sd x30, 0(x31)								IF	○	ID	EX	MEM	WB							
bne x12, x13, TOP								IF	○	ID	EX	MEM	WB							
nop																				
TOP: slli x5, x12, 3										IF	ID	EX	MEM	WB						
nop																				
add x6, x10, x5											IF	ID	EX	MEM	WB					
ld x7, 0(x6)												IF	ID	EX	MEM	WB				
add x31, x11, x5												IF	ID	EX	MEM	WB				
ld x29, 8(x6)													IF	ID	EX	MEM	WB			
addi x12, x12, 2													IF	ID	EX	MEM	WB			
nop																				
sub x30, x7, x29														IF	ID	○	EX	MEM	WB	
sd x30, 0(x31)															IF	○	ID	EX	MEM	WB
bne x12, x13, TOP															IF	○	ID	EX	MEM	WB
finish																				

4.31.6. In 4.31.3, each iteration needs 9 cycles. In 4.31.4, each iteration needs 6 cycles.
So the speedup is $\frac{9}{6} = 1.5$

4.31.7.

```
      beqz   x13, DONE
      li     x12, 0
TOP:  slli   x5, x12, 3
      add    x6, x10, x5
      add    x31, x11, x5
      ld     x7, 0(x6)
      ld     x29, 8(x6)
      ld     x28, 16(x6)
      ld     x30, 24(x6)
      addi   x12, x12, 4
      sub    x29, x7, x29
      sub    x30, x28, x30
      sd     x29, 0(x31)
      sd     x30, 16(x31)
      bne    x12, x13, TOP
DONE:
```

4.31.8.

```
      beqz   x13, DONE
      li     x12, 0
      addi   x6, x10, 0
TOP:  ld     x7, 0(x6)
      slli   x5, x12, 3
      ld     x29, 8(x6)
      add    x31, x11, x5
      ld     x28, 16(x6)
      addi   x12, x12, 4
      ld     x30, 24(x6)
      sub    x29, x7, x29
      sd     x29, 0(x31)
      sub    x30, x28, x30
      sd     x30, 16(x31)
      addi   x6, x10, 32
      bne    x12, x13, TOP
DONE:
```

4.31.9. In 4.31.7, each iteration needs 13 cycles. In 4.31.8, each iteration needs 7 cycles.
So the speedup is $\frac{13}{7} \fallingdotseq 1.85$

4.31.10. It is the same as 4.31.8 except that two consecutive instructions among the three instructions (beqz, li, addi) can be combined into a packet. However, it does not reduce the needed cycle per iteration, so the speedup is the same.

5.5.1. offset is 4~0 ⇒ block size is $2^5 = 32$, assume that each word is 8 bytes, block size = $\frac{32}{8} = 4$ words

5.5.2. Index is 9~5 ⇒ there $2^5 = 32$ blocks

5.5.3. For data storage: $32 \cdot 32 \cdot 8$

Total required : $32 \cdot (32 \times 8 + \overset{\text{tag}}{54} + \overset{\text{valid bit}}{1})$ ⇒ $\frac{32 \times (32 \times 8 + 54 + 1)}{32 \times 32 \times 8} \fallingdotseq 1.21$

5.5.4.

Address	Tag	Index	Offset	Hit/Miss	Replaced
0x00	0	0	0	Miss	
0x04	0	0	4	Hit	
0x10	0	0	16	Hit	
0x84	0	4	4	Miss	
0xE8	0	7	8	Miss	
0xA0	0	5	0	Miss	
0x400	1	0	0	Miss	0x00~0x1F
0x1E	0	0	31	Miss	0x400~0x41F
0x8C	0	4	12	Hit	
0xC1C	3	0	28	Miss	0x00~0x1F
0xB4	0	5	20	Hit	
0x884	2	4	4	Miss	0x80~0x9F

The Address and Replaced columns are in heximal, and other columns are in decimal.

5.5.5. Hit ratio = $\frac{4}{12} \fallingdotseq 0.33$

5.5.6. < Index, Tag, Data >

< 0 , 3 , Mem[0xC00] - Mem[0xC1F] >

< 4 , 2 , Mem[0x880] - Mem[0x89F] >

< 5 , 0 , Mem[0xA0] - Mem[0xBF] >

< 7 , 0 , Mem[0xE0] - Mem[0xFF] >

5.10.1. P1: $\frac{1}{0.66 \times 10^{-9}} \fallingdotseq 1.515 \times 10^{9}$ Hz, P2. $\frac{1}{0.9 \times 10^{-9}} \fallingdotseq 1.11 \times 10^{9}$ Hz

5.10.2. P1: $1 + 0.08 \times \lceil \frac{70 \times 10^{-9}}{0.66 \times 10^{-9}} \rceil = 1 + 0.08 \times 107 = 9.56$ cycles

P2: $1 + 0.06 \times \lceil \frac{70 \times 10^{-9}}{0.9 \times 10^{-9}} \rceil = 1 + 0.06 \times 78 = 5.68$ cycles

5.10.3. P1: $1 + 0.08 \times 107 + 0.36 \times 0.08 \times 107 = 12.64$ cycles

$12.64 \times 0.66 \times 10^{-9} = 8.34 \times 10^{-9}$ s.

P2. $1 + 0.06 \times 78 + 0.36 \times 0.06 \times 78 = 7.36$ cycles

$7.36 \times 0.9 \times 10^{-9} = 6.62 \times 10^{-9}$ s

⇒ P2 is faster

5.10.4. $\text{AMAT} = 1 + 0.08 \times \lceil \frac{5.62 \times 10^{-9}}{0.66 \times 10^{-9}} \rceil + 0.08 \times 0.95 \times \lceil \frac{70 \times 10^{-9}}{0.66 \times 10^{-9}} \rceil$

$= 1 + 0.08 \times 9 + 0.08 \times 0.95 \times 107$

$= 9.85$ cycles ⇒ worse with the L2 cache

3.10.5. $\text{CPI} = 1 + (\text{AMAT} - 1) + 0.36 \times (\text{AMAT} - 1) \fallingdotseq 13.04$ cycles

Let $\text{AMAT}' = 1 + 0.08 \times 9 + 0.08 \times r \times 107$. r is the miss rate of L2 cache

$\text{CPI}' = 1.36 \cdot \text{AMAT}' - 0.36$

5.10.6. $(1.36 \times \text{AMAT}' - 0.36) \times 0.66 \times 10^{-9} < 9.85 \times 0.66 \times 10^{-9}$

$r < 0.915$

5.10.7. $(1.36 \times \text{AMAT}' - 0.36) \times 0.66 \times 10^{-9} < 7.36 \times 0.9 \times 10^{-9}$

$r < 0.692$

5.16.1

4KB page => the last 12bits of address are for offset

Address	Page Table Index	TLB Hit/Miss	Page Table Hit/Miss	Page Fault	TLB			
					Valid	Tag	Physical Page Number	Time Since Last Access
0x123d	1	Miss	Hit	Yes	1	0xb	12	5
					1	0x7	4	2
					1	0x3	6	4
					1	0x1	13	0
0x08b3	0	Miss	Hit	No	1	0x0	5	0
					1	0x7	4	3
					1	0x3	6	5
					1	0x1	13	1
0x365c	3	Hit	Hit	No	1	0x0	5	1
					1	0x7	4	4
					1	0x3	6	0
					1	0x1	13	2
0x871b	8	Miss	Hit	Yes	1	0x0	5	2
					1	0x8	14	0
					1	0x3	6	1
					1	0x1	13	3
0xbee6	b	Miss	Hit	No	1	0x0	5	3
					1	0x8	14	1
					1	0x3	6	2
					1	0xb	12	0
0x3140	3	Hit	Hit	No	1	0x0	5	4
					1	0x8	14	2
					1	0x3	6	0
					1	0xb	12	1
0xc049	c	Miss	Hit	Yes	1	0xc	15	0
					1	0x8	14	3
					1	0x3	6	1
					1	0xb	12	2

5.16.2

16KB page => the last 14bits of address are for offset

Address	Page Table Index	TLB Hit/Miss	Page Table Hit/Miss	Page Fault	TLB			
					Valid	Tag	Physical Page Number	Time Since Last Access
0x123d	0	Miss	Hit	No	1	0xb	12	5
					1	0x7	4	2
					1	0x3	6	4
					1	0x0	5	0
0x08b3	0	Hit	Hit	No	1	0xb	12	6
					1	0x7	4	3
					1	0x3	6	5
					1	0x0	5	0
0x365c	0	Hit	Hit	No	1	0xb	12	7
					1	0x7	4	4
					1	0x3	6	5
					1	0x0	5	0
0x871b	2	Miss	Hit	Yes	1	0x2	13	0
					1	0x7	4	5
					1	0x3	6	6
					1	0x0	5	1
0xbee6	2	Hit	Hit	No	1	0x2	13	0
					1	0x7	4	6
					1	0x3	6	7
					1	0x0	5	2
0x3140	0	Hit	Hit	No	1	0x2	13	1
					1	0x7	4	7
					1	0x3	6	8
					1	0x0	5	0
0xc049	3	Hit	Hit	Yes	1	0x2	13	2
					1	0x7	4	8
					1	0x3	6	0
					1	0x0	5	1

Advantages : increase the TLB hit ratio, decrease the size of page table

Disadvantages : need more time to do swapping, increase internal fragmentation

5.16.3

4KB page => the last 12bits of address are for offset

Assume that the TLB index is determined by Page Table Index module 2.

Because in the initial state of TLB, 0xb and 0x7 should be in index 1 entries but in index 0 entries, I set the valid bits of these two entries to 0.

Address	Page Table Index	TLB Hit/Miss	Page Table Hit/Miss	Page Fault	TLB				
					Index	Valid	Tag	Physical Page Number	Time Since Last Access
0x123d	1	Miss	Hit	Yes	0	0	0xb	12	5
					0	0	0x7	4	2
					1	1	0x3	6	4
					1	1	0x1	13	0
0x08b3	0	Miss	Hit	No	0	1	0x0	5	0
					0	0	0x7	4	3
					1	1	0x3	6	5
					1	1	0x1	13	1
0x365c	3	Hit	Hit	No	0	1	0x0	5	1
					0	0	0x7	4	4
					1	1	0x3	6	0
					1	1	0x1	13	2
0x871b	8	Miss	Hit	Yes	0	1	0x0	5	2
					0	1	0x8	14	0
					1	1	0x3	6	1
					1	1	0x1	13	3
0xbee6	b	Miss	Hit	No	0	1	0x0	5	3
					0	1	0x8	14	1
					1	1	0x3	6	2
					1	1	0xb	12	0
0x3140	3	Hit	Hit	No	0	1	0x0	5	4
					0	1	0x8	14	2
					1	1	0x3	6	0
					1	1	0xb	12	1
0xc049	c	Miss	Hit	Yes	0	1	0xc	15	0
					0	1	0x8	14	3
					1	1	0x3	6	1
					1	1	0xb	12	2

5.16.4

4KB page => the last 12bits of address are for offset

Assume that the TLB index is determined by Page Table Index module 4.

Because in the initial state of TLB, the entries are not consistent with my assumption, I set the valid bits of these entries to 0.

Address	Page Table Index	TLB Hit/Miss	Page Table Hit/Miss	Page Fault	TLB				
					Index	Valid	Tag	Physical Page Number	Time Since Last Access
0x123d	1	Miss	Hit	Yes	0	0	0xb	12	5
					1	1	0x1	13	0
					2	0	0x3	6	4
					3	0	0x4	9	7
0x08b3	0	Miss	Hit	No	0	1	0x0	5	0
					1	1	0x1	13	1
					2	0	0x3	6	5
					3	0	0x4	9	8
0x365c	3	Miss	Hit	No	0	1	0x0	5	1
					1	1	0x1	13	2
					2	0	0x3	6	6
					3	1	0x3	6	0
0x871b	8	Miss	Hit	Yes	0	1	0x8	14	0
					1	1	0x1	13	3
					2	0	0x3	6	7
					3	1	0x3	6	1
0xbee6	b	Miss	Hit	No	0	1	0x8	14	1
					1	1	0x1	13	4
					2	0	0x3	6	8
					3	1	0xb	12	0
0x3140	3	Miss	Hit	No	0	1	0x8	14	2
					1	1	0x1	13	5
					2	0	0x3	6	9
					3	1	0x3	6	0
0xc049	c	Miss	Hit	Yes	0	1	0xc	15	0
					1	1	0x1	13	6
					2	0	0x3	6	10
					3	1	0x3	6	1

5.16.5

If there is n TLB, then each memory access will have to access to memory two times. The first is to access to page table to get the physical page number. The second is to access to that page and get data.

So a high performance CPU should have a TLB to reduce memory access time.

6.7.1.

(x, y, w, z)	Execution order
(2, 2, 1, 0)	3 → 4 → 1 → 2
(2, 2, 1, 2)	3 → 1 → 4 → 2
(2, 2, 1, 4)	3 → 1 → 2 → 4
(2, 2, 3, 0)	4 → 1 → 3 → 2
(2, 2, 3, 2)	1 → 4 → 3 → 2
(2, 2, 3, 4)	1 → 3 → 2 → 4
(2, 2, 5, 0)	4 → 1 → 2 → 3
(2, 2, 5, 2)	1 → 4 → 2 → 3
(2, 2, 5, 4)	1 → 2 → 4 → 3

6.7.2. Use synchronization instructions after changing value of variables so that other cores can be aware of the new value of variables.

6.9.1.

Core 1	Core 2
A3. A2	B4. B2
A1. A4	B4. B3
A1.	B1
A1	B1

⇒ 4 cycles, 4 slots are wasted

6.9.2.

CPU 1		CPU 2	
Core 1	Core 2	Core 1	Core 2
A1. A2	B1.	A3	B4. B2
A1	B1	A4	B4. B3
A1			

⇒ 3 cycles, 12 slots are wasted

6.9.3.

FU1	FU2
A1	A2
A1	
A1	
A3	
A4	
B1	
B1	
B2	
B3	B4
	B4

⇒ 10 cycles, 8 slots are wasted

6.9.4.

FU1	FU2
A1	B1
A1	B1
A1	B2
A2	B3
A3	B4
A4	B4

⇒ 6 cycles, no slot is wasted

Programming Part 1:

	dhrystone	median	multiply	qsort	rsort	towers	vvadd
Config 1	557936	8863	44964	269251	900737	7497	11830
Config 2	539075	8817	44947	257841	902477	7497	5053
Config 3	542214	8881	45032	257034	911861	7577	4808
Config 4	545513	8864	45111	254099	884849	7577	4653
Config 5	527386	8864	45112	254384	885937	7577	4653
Config 6	574790	8789	44900	269251	901048	7457	11830
Config 7	582962	8789	44892	269342	900876	7476	11808
Config 8	551369	9337	45091	274111	1025081	7485	12795
Config 9	551704	9315	45096	274363	1026321	7485	12872
Config 10	552352	9292	45101	274172	1026003	7499	13006
Config 11	546999	9390	45127	275235	1031835	7501	12648
Config 12	549202	9330	45112	263335	1051311	7606	5476
Config 13	547675	9361	45244	263814	1051300	7599	5541

(1) Green : Different(11830 and 5053).

The vvadd benchmark does matrix addition c[i] = a[i] + b[i].

Config 1 is L1_Dcache_1-way, so there is only one entry in a set to store data. Because data is read in the pattern array_a -> array_b -> array_a ..., this will cause data in Dcache be overwritten again and again.

Config 2 is L1_Dcache_2-way, so there are two entries in a set to store data of array_a and array_b respectively. Thus, data in Dcache will not be overwritten so frequently.

The difference in cache entries in a set leads to the difference in cycle count.

(2) Red : Different(911861 and 884849).

The rsort benchmark does radix sort.

Config 3 use random replacement, while Config 4 use LRU replacement.

Because radix sort read array data orderly, using LRU as replacement policy is more like the program’s access pattern than using random replacement. Thus Config 4 has smaller cycle count.

(3) Blue : Different(900737 and 911861).

The rsort benchmark does radix sort.

Config 1 is L1_Dcache_1-way, while Config 3 is L1_Dcache_2-way.

Because the program only reads data orderly in a single array, there is no need for multiple cache entries in a set. Besides, more cache entries in a set will slow down the speed of cache, so Config 1 with only one cache entry in a set has smaller cycle count.

(4) Yellow : Different(557936 and 574790 and 582962).
Config 1 has 1-way Icache, Config 6 has 2-way Icache, Config has 4-way Icache.
The program run several specific instruction blocks for many times, and smaller entries number in a set can reduce the time needed to fetch instruction, so the cycle count is Config 1 < Config 6 < Config 7.

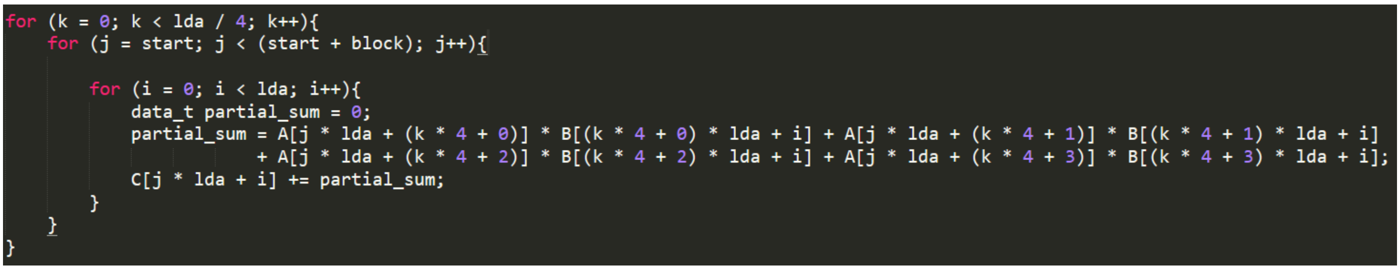
(5) Brown : Different(549202 and 547675).
Config 12 has 1-bank L2 cache, while Config 13 has 4-bank L2 cache.
Config 13 has 4 banks, thus increase parallelism, and it can improve bandwidth and reduce cycle count. So the cycle count is Config 13 < Config 12.

(6) pmp.c :
It wants to test whether the Physical Memory Protection functionality works.
It does this by testing whether each memory address is accessible by calling exhaustive_test and test_range. The program will return 0 if the functionality works well.

(7) Config17 on 1-core : 180005 cycles
Config19 on 2-core : 92287 cycles
Config20 on 4-core : 48239 cycles
The cycle count decrease linearly approximately. Because the task that the program does can be divided into several equal parts and thus can run on multiple cores simultaneously. So the cycle count can decrease linearly when the core number increase.

Programming Part2 Report :

Code :



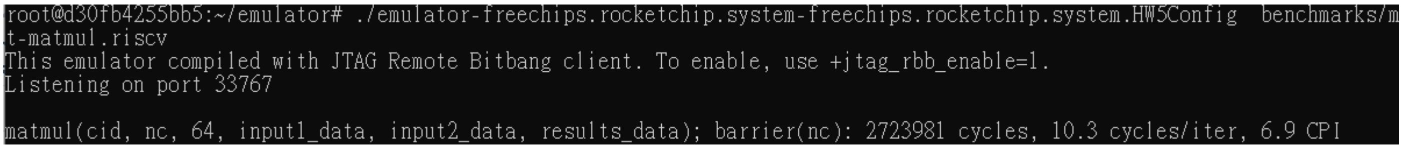
```
for (k = 0; k < lda / 4; k++){
    for (j = start; j < (start + block); j++){

        for (i = 0; i < lda; i++){
            data_t partial_sum = 0;
            partial_sum = A[j * lda + (k * 4 + 0)] * B[(k * 4 + 0) * lda + i] + A[j * lda + (k * 4 + 1)] * B[(k * 4 + 1) * lda + i]
                        + A[j * lda + (k * 4 + 2)] * B[(k * 4 + 2) * lda + i] + A[j * lda + (k * 4 + 3)] * B[(k * 4 + 3) * lda + i];
            C[j * lda + i] += partial_sum;
        }
    }
}
```

I set L1_Dcache_Set = 2 and L1_Dcache_Way = 8 and L1_Dcache_replacement = plru.

There are four cores to do matrix multiplication, so I divide the task into four equal parts to each core.

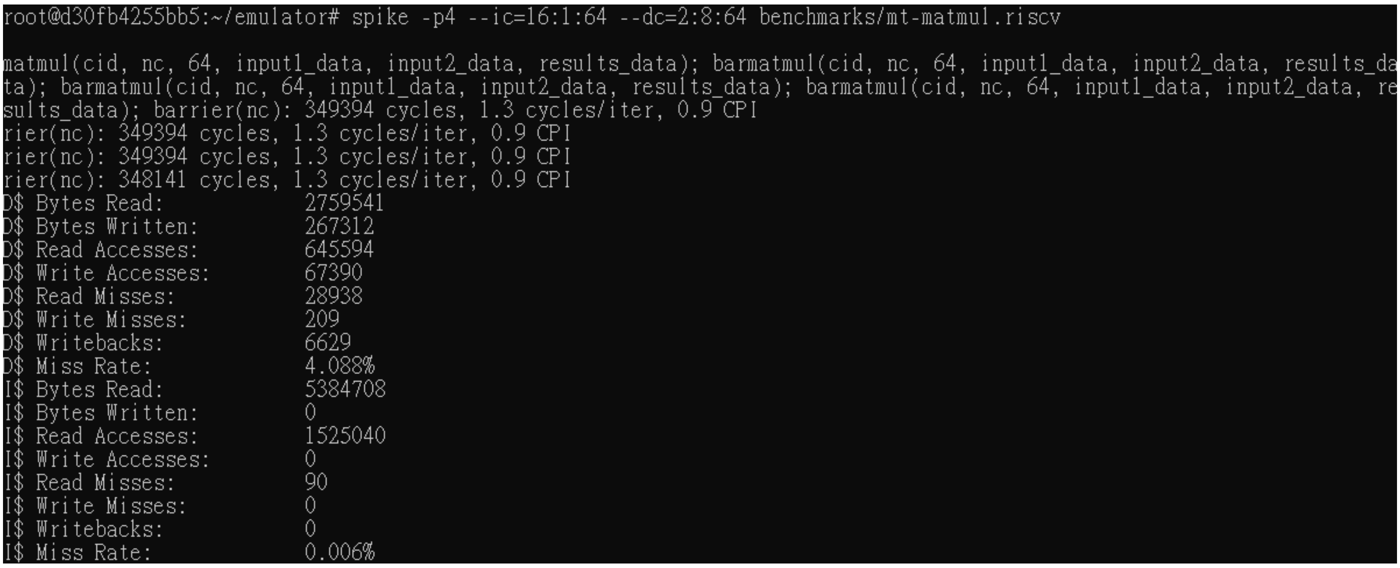
During the matrix multiplication, I let the cached data can be used as many as possible to reduce cycles by modifying original code to above.

Result :



```
root@d30fb4255bb5:~/emulator# ./emulator-freechips.rocketchip.system-freechips.rocketchip.system.HW5Config  benchmarks/m
t-matmul.riscv
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 33767

matmul(cid, nc, 64, input1_data, input2_data, results_data); barrier(nc): 2723981 cycles, 10.3 cycles/iter, 6.9 CPI
```

After modifying code and HW5Config, the result is shown above. Cycle count is 2723981.



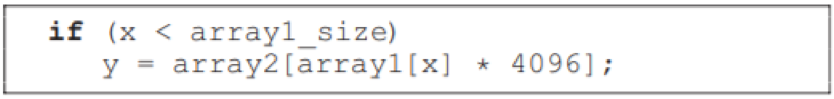
```
root@d30fb4255bb5:~/emulator# spike -p4 --ic=16:1:64 --dc=2:8:64 benchmarks/mt-matmul.riscv

matmul(cid, nc, 64, input1_data, input2_data, results_data); barmatmul(cid, nc, 64, input1_data, input2_data, results_da
ta); barmatmul(cid, nc, 64, input1_data, input2_data, results_data); barmatmul(cid, nc, 64, input1_data, input2_data, re
sults_data); barrier(nc): 349394 cycles, 1.3 cycles/iter, 0.9 CPI
rier(nc): 349394 cycles, 1.3 cycles/iter, 0.9 CPI
rier(nc): 349394 cycles, 1.3 cycles/iter, 0.9 CPI
rier(nc): 348141 cycles, 1.3 cycles/iter, 0.9 CPI
D$ Bytes Read:             2759541
D$ Bytes Written:          267312
D$ Read Accesses:          645594
D$ Write Accesses:         67390
D$ Read Misses:            28938
D$ Write Misses:           209
D$ Writebacks:             6629
D$ Miss Rate:              4.088%
I$ Bytes Read:             5384708
I$ Bytes Written:          0
I$ Read Accesses:          1525040
I$ Write Accesses:         0
I$ Read Misses:            90
I$ Write Misses:           0
I$ Writebacks:             0
I$ Miss Rate:              0.006%
```

Using spike to find Dcache miss rate. The Dcache miss rate under this setting is 4.088%.

Bonus : Architecture and Security

Exploiting conditional branch misprediction attack :

由於 CPU 為了提高速度而會在 branch 條件確認前先執行指令，攻擊者能知道某個 byte 在記憶體的哪個位置。



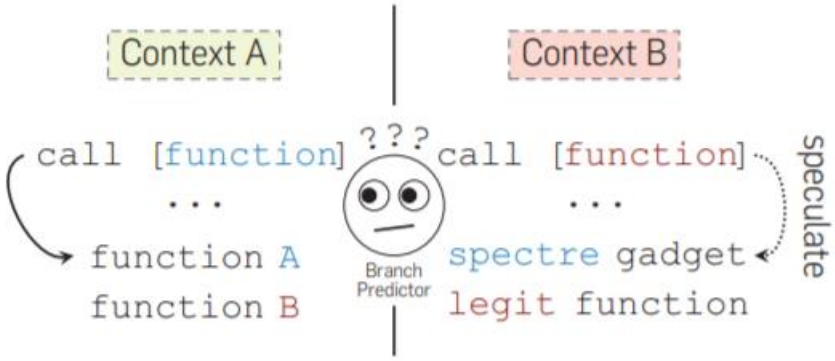
```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

假設：設計 x 的值使 x 超出 array1_size 的範圍且 array1[x]會決定某個 byte k 在記憶體中的位置
array1_size 和 array2 不在 cache 中，但 byte k 在 cache 中
先前的 x 值都是合法的，導致 branch predictor 認為 if 條件可能為真

攻擊者挑選 x 的值，而 CPU 因為 array1_size 不在 cache 中，因此會在確認 if 條件之前就先假設條件為真而繼續往下執行，接著向記憶體要 array1[x]的資料，而因為 k=array1[x]在 cache 中，因此會很快回傳，接著向記憶體要 array2[k*4096]的資料，但此時由於 cache miss，因此不會馬上回傳。當 CPU 知道條件錯誤並回溯暫存器的狀態時，cache 的狀態卻仍被錯誤要到的 array2 的資料改變。
攻擊者接著測量 array2 中哪一塊區域由於在 cache 中，因此回傳得特別快，就能知道 byte k 的值，完成攻擊。

Poisoning indirect branches attack :

攻擊者首先利用自己的程式誤導 CPU 的 branch predictor，使其在執行其他程式時是依照執行攻擊者程式時的 branch 猜測依據，導致在執行其他程式時，CPU 可能會執行到不應該被執行到的程式部分。



如圖所示，攻擊者藉由程式 A 訓練（誤導）branch predictor，使其在執行程式 B 時，當要進行 branch 時，會猜測 branch destination 仍和程式 A 相同，使攻擊成立。

Mitigate Spectre Attacks :

1. 不讓 CPU 做 speculative execution
   雖然能抵擋 spectre attacks，但會使 CPU 執行效率降低
2. 不使用 speculative execution 執行錯誤所帶進來的資料
   但目前的 CPU 仍不具備這個功能，或許未來的 CPU 有可能有能力分辨資料來源
3. 防止 branch poisoning
   Intel 和 AMD 增加他們的 ISA 及防護機制來限制攻擊者影響 branch speculation 的能力