

# CSE 351 Final Exam

Winter 2019

Tuesday, March 19, 2019

Name: \_\_\_\_\_

UW NetID: \_\_\_\_\_

## Instructions

- Show scratch work for partial credit, but put your final answers in the blanks provided.
- Write your UW NetID on the top right corner of **every page**.
- The last sheet is a reference sheet. **Please detach it from the rest of the exam.**
- You have 1 hour and 50 minutes for the exam.
- There are 210 points, plus 10 extra credit points, for a total of 220 points.
- **This is a long exam.** We are giving you many opportunities to show what you know and earn points. If you can't get something, relax. Do not get stuck, **move on to another problem.**

Question	Points	Score
Arrays and Structs	30	
x86-64 Assembly	16	
Procedures	20	
Buffer Overflow	22	
Processes	20	
Caches	32	
Virtual Memory	30	
Allocation	20	
Memory Bugs	20	
<i>Extra Credit</i>	10	
Total:	220	

**Question 1: Arrays and Structs****(30 total points)**

The following questions refer to this definition of `cse_building`.

```
typedef struct {
    int num_donors;           // number of contributors
    char** donor_names;      // pointer to strings representing donors
    char has_soft_seats;     // 1 if has soft furniture, 0 otherwise
    char location_name[10];  // C string for building location
    short height;           // height, in feet
} cse_building;
```

(a) Give the following quantities as numbers of bytes in the blanks on the right.

i. (2 points) `sizeof(cse_building)` \_\_\_\_\_

ii. (2 points) Internal fragmentation of `cse_building` \_\_\_\_\_

iii. (2 points) External fragmentation of `cse_building` \_\_\_\_\_

(b) Given the following declaration:

```
cse_building building;
```

Rewrite the following expressions using only pointer arithmetic, `&`, `*`, and casting (so no dot or arrow operators).

i. (2 points) `building.num_donors`

ii. (2 points) `&building.donor_names`

- (c) (8 points) Implement the following function by code below that would go in the designated loop body.

You may assume that `cse_building` always has non-negative `num_donors`.

```
// params:
//     a -> pointer to the beginning of an array of cse_buildings
//     len -> number of cse_buildings in the array (> 0)
// returns:
//     a pointer to the cse_building with the most donors
//     (if it's a tie, return the first one in the array).
cse_building* has_most_donors(cse_building* a, int len) {
    cse_building* max_donors;
    int max_donor_num = 0;
    for (int i = 0; i < len; i++) {
        // ----- //
        // WRITE CODE THAT GOES HERE //
        // ----- //
    }
    return max_donors;
}
```

- (d) (4 points) Your friend (who also took CSE 351) claims to have a more space efficient way of storing this information, showing you the following:

```
typedef struct {
    char** donor_names;      // pointer to strings representing donors
    short height;           // height, in feet
    int num_donors;         // number of contributors
    char has_soft_seats;    // 1 if has soft furniture, 0 otherwise
    char location_name[10]; // C string for building location
} cse_building_better;
```

Are they right in saying it's more space efficient? Why or why not?

- (e) (8 points) Suppose that we knew that there can be at most 3 donors for any `cse_building`, and that their names would be no longer than 13 characters (plus one null terminator). We can use this information to fill in the following struct definition:

```
typedef struct {
    char donor_names[3][14];
    short height;
    int num_donors;
    char has_soft_seats;
    char location_name[10];
} cse_building_v2;
```

Name 1 benefit of this alternate representation over the one shown in part (d). Name 1 drawback as well.

**Question 2: x86-64 Assembly****(16 total points)**

(a) `pushq` and `popq` are important instructions for manipulating the stack. However, they can be replaced by combinations of other instructions.

i. (4 points) Write a sequence of assembly instruction(s) without a `pushq` that has the same behavior as `pushq %rax`.

ii. (4 points) Write a sequence of assembly instruction(s) without a `popq` that has the same behavior as `popq %rax`.

(b) (4 points) What is the difference between `movq $1, %rax` and `movq $1, (%rax)`? (1 sentence)

(c) (4 points) Given the following (decimal) values of registers:

`%rax` = 20

`%rbx` = 5

What is the value of `%rsi` after the following instruction:

```
leaq 3(%rax, %rbx, 2), %rsi
```

**Question 3: Procedures****(20 total points)**

Consider the following C program fragment compiled without and with a common optimization called *inlining*. The left side shows the unoptimized C and corresponding assembly. The right side shows the assembly compiled with inlining, and above that is “manually” inlined C code.

To inline a function, the compiler essentially replaces calls to that function with the whole body of the function. In the below optimized code on the right, `times` has been inlined into the body of `cube`.

<pre>// the initial C code <b>int</b> times(<b>int</b> x, <b>int</b> y) {     <b>return</b> x * y; }  <b>int</b> cube(<b>int</b> x) {     <b>int</b> x2 = times(x, x);     <b>return</b> times(x, x2); }  // unoptimized assembly times:     movl    %edi, %eax     imull  %esi, %eax     ret cube:     pushq  %rbx     movl   %edi, %ebx     movl   %edi, %esi     call  times     movl   %eax, %esi     movl   %ebx, %edi     call  times     popq   %rbx     ret</pre>	<pre>// the manually optimized C <b>int</b> times(<b>int</b> x, <b>int</b> y) {     <b>return</b> x * y; }  <b>int</b> cube(<b>int</b> x) {     <b>int</b> x2 = x * x;     <b>return</b> x * x2; }  // optimized assembly times:     movl    %edi, %eax     imull  %esi, %eax     ret cube:     movl    %edi, %eax     imull  %edi, %eax     imull  %edi, %eax     ret</pre>
---	--

(a) i. (2 points) Which callee-saved registers (if any) does the unoptimized `cube` use?

ii. (2 points) Which callee-saved registers (if any) does the optimized `cube` use?

The optimization makes the code run 4 times faster! You can see that the optimized `cube` has no `call` instructions anymore, instead it has two `imulls`. But it also has far fewer total instructions.

- (b) (6 points) Explain briefly why the compiler was able to remove the `pushq` and `popq` instructions and many of the move instructions

While it's one of the most important optimizations performed by compilers, inlining is not always effective or even possible. Now we will consider some of the limitations of this technique.

- (c) (5 points) Consider recursive functions. Can a recursive function be inlined into itself? If so, are there limitations or drawbacks? If not, explain why. Your response should be 2-3 concise sentences.

- (d) (5 points) Consider functions that are very large (in terms of number of instructions) or take a long time to run (have long running loops). These functions *can* be inlined into other functions, but the benefit is much smaller than we saw in the example on the previous page. In some cases, inlining these can make your code slower!

Why is the benefit smaller? Your response should be 2-3 concise sentences.

**Question 4: Buffer Overflow****(22 total points)**

You're Sokka — fighting the good fight against the invading Fire Nation army. To stop a legion of Fire Nation ships, you've decided to hack into them and remotely disable them!

During your mission, you've learned that their control system runs on a system very similar to x86-64 Linux, but with two key differences:

- **Addresses are 32-bits (and thus so are pointers)**
- **All arguments to functions are passed on the stack**

You've also obtained an excerpt on their control system's source code:

```

1 void set_all_ship_controls(char* mode);
2 void check_password(char* password);
3
4 // pass_input_to_function's parameter is a function pointer,
5 // which is the address of a function
6 void pass_input_to_function(void (*f)(char*)) {
7     char input[16];
8     gets(input);
9     f(input); // Calls f based on the argument stored on the stack
10 }
11
12 void login() {
13     printf("Input your password:\n");
14     pass_input_to_function(&check_password);
15     printf("Done!\n");
16 }

```

Recall that `gets()` is a libc function that reads characters from standard input until the newline (`'\n'`) character is encountered. The resulting characters (not including the newline) are stored in the buffer that's given to `gets()` as a parameter. If any characters are read, `gets()` appends a null-terminating character (`'\0'`) to the end of the string.

The normal behavior of the above program is as follows. Once `login` is called, it will print "Input your password:". Then it will call `pass_input_to_function` with a pointer to the function `check_password` as the argument. `pass_input_to_function` calls `gets` to get input from `stdin`, then it calls the given function (in this case, `check_password`) with the input received from `gets` as the argument. Then it returns, and then `login` prints "Done!" and returns.

- (a) (4 points) Explain why the use of the `gets()` function introduces a security vulnerability in the program?



The table below shows the stack right before line 9 (from the previous page) runs. The stack pointer is `0x7fff8000`.

Address	Value
<code>0x7fff800C</code>	<code>&amp;f</code>
<code>0x7fff8008</code>	return address to line 15
<code>0x7fff8004</code>	
<code>0x7fff8000</code>	start of buffer input

- (b) (6 points) You've gained access to their control system, but you don't know the password. Thankfully, you've learned that `set_all_ship_controls` is located at address `0x351`, and that calling `set_all_ship_controls("off!")` will cause all the ships to stop working!

Construct a sequence of bytes that can get sent to the program during a call to `login()` (read by `gets`) and trigger the above vulnerability and cause all the ships to stop working. Set any other overwritten stack values to 0; ignore any problems this may cause for now.

**Give your answer as a sequence of bytes, each written in hex.**

Put space between each byte (like this: "13 AE 00 00 4E 00 00 00 00").

*Hint.* The ASCII values (in hex) of the characters in "off!" are: `6f 66 66 21`

- (c) (4 points) You notice that, after constructing the string above, there is one more issue: the program will segfault at some point after calling `set_all_ship_controls`! What is the cause of this crash, and how could we change our attack string to prevent it? Explain in 1-2 sentences.

(d) (4 points) How could `pass_input_to_function` be modified to fix the vulnerability identified in part (a)? Explain in 1 sentence.

(e) (4 points) Describe two ways the compiler, hardware, or operating system (on regular x86-64 Linux machines) protect against buffer overflow attacks.

## Question 5: Processes

(20 total points)

(a) Consider the following function that uses Linux's fork, wait, and exit.

```
1 void fun_with_process() {
2     int status;
3     int x = 2;
4
5     if (fork() == 0) {
6         if (fork() == 0) {
7             x += 3;
8             printf(" %d ", x);
9             exit(0);
10        } else {
11            wait(&status);
12            x--;
13            printf(" %d ", x);
14        }
15    } else {
16        x++;
17    }
18    printf(" %d ", x);
19    exit(0);
20 }
```

i. (2 points) How many different processes will execute line 18?

ii. (8 points) What are all the different possible outputs (i.e. order of things printed) for this code?

- (b) Recall that a context switch is when the operating system switches which process is executing. For each of the following items, choose how frequently they would be the same before and after a context switch. In other words, will they be the same in the old running process and the new one? Assume that the operating system will choose a different process from the currently running process upon a context switch, and that the processes do not share any virtual to physical page mappings.
- i. (2 points) The running process ID
    - Always
    - Sometimes
    - Never
  
  - ii. (2 points) The valid entries in the TLB
    - Always
    - Sometimes
    - Never
  
  - iii. (2 points) The contents of the page table base register
    - Always
    - Sometimes
    - Never
  
  - iv. (2 points) The contents of the L1 (physical) cache
    - Always
    - Sometimes
    - Never
  
  - v. (2 points) The program executed by the process
    - Always
    - Sometimes
    - Never

**Question 6: Caches****(32 total points)**

(a) (4 points) What is the difference between spatial and temporal locality? Answer in 1-2 sentences.

(b) (4 points) How can a write-back cache improve the performance of writes? Explain in 1-2 sentences.

(c) For each of the following sequences of memory accesses, determine the best cache parameters to reduce miss rate. Some parameters will be given, so fill in the remaining one.

Assume that the cache is empty at the beginning of each sequence, the cache uses an LRU replacement policy, and that all accesses are valid.

i. (3 points) 0x00, 0x08, 0x10, 0x18, 0x20, 0x28, 0x30, 0x38

Total cache size = 32 bytes

Associativity = 1

Block size = \_\_\_\_\_

ii. (3 points) 0x00, 0x20, 0x40, 0x80, 0x01, 0x21, 0x41, 0x81

Total cache size = 32 bytes

Block size = 8 bytes

Associativity = \_\_\_\_\_

iii. (3 points) 0x00, 0x01, 0x38, 0x48, 0x00, 0x01

Total cache size = 16 bytes

Block size = 8 bytes

Associativity = \_\_\_\_\_



**Question 7: Virtual Memory****(30 total points)**

- (a) Virtual memory is an extremely powerful abstraction with many benefits. For each benefit listed below, provide a brief (1-2 sentence) explanation of how VM accomplishes it.
- i. (4 points) Protecting processes from one another.
  
  
  
  
  
  
  
  
  
  
  - ii. (4 points) Allow programs to use more memory than exists on the machine.
- (b) Last month, I turned off my desktop computer, opened it up, added some more RAM (doubling it from 16GiB to 32GiB), closed it, and turned it back on.
- i. (4 points) Did the size of the virtual address space change? Why or why not?
  
  
  
  
  
  
  
  
  
  
  - ii. (4 points) Did the size of the physical address space change? Why or why not?
  
  
  
  
  
  
  
  
  
  
  - iii. (4 points) After the RAM upgrade, I was able to have more programs open (and thus using memory) before performance began to degrade. Why did this happen? What was causing performance to drop after a certain number of programs were running? (1-2 sentences)

(c) Imagine the following system:

- 16-bit virtual addresses, 10-bit physical addresses.
- A page size of 16 bytes.
- 2-way set associative TLB with 8 total entries.
- All page table entries NOT in the initial TLB start as invalid.

i. (4 points) Compute the following quantities

Page offset bits: \_\_\_\_\_ PPN bits: \_\_\_\_\_

VPN bits: \_\_\_\_\_ TLB index bits: \_\_\_\_\_

ii. (6 points) The TLB has the following state:

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	0x1B2	-	0	0x283	0x3A	1
1	0x2FB	0x29	1	0x0E8	0x1D	1
2	0x004	-	0	0x346	-	0
3	0x3F4	0x1B	1	0x257	0x36	1

Fill in the associated information for the following accesses to virtual addresses. (enter **n/a** if the answer cannot be determined):

Virtual Address	TLB Hit?	Page Fault?	PPN
0x3A17	_____	_____	_____
0x0123	_____	_____	_____



## Question 8: Allocation

(20 total points)

- (a) (5 points) In some versions of C, you can make dynamically-sized allocations on the stack. In the listing below, the array `on_the_stack` of ints is stored on the stack inside `foo`'s stack frame.

```
int foo(unsigned n) {
    int on_the_stack[n];
    for (int i = 0; i < n; i++) {
        on_the_stack[i] = 0;
    }
    // do something with on_the_stack
}
```

Does this replace the need for `malloc`? Explain in 1-2 sentences.

- (b) (5 points) Suppose I write and use the following code in a version of C with a mark-and-sweep garbage collector.

```
long do_it(int* p) {
    return ~(long)p;
}

int* undo_it(long l) {
    return (int*)(~p);
}

void bad_stuff() {
    long stash = do_it(malloc(sizeof(int)));

    // run a bunch of code

    int *p = undo_it(stash);
    *p = 0xC0FFEE;
}
```

In 1-2 sentences, explain how this is potentially dangerous.

(c) i. (4 points) Describe 2 advantages of an implicit free list allocator over an explicit list allocator. (1-2 sentences)

ii. (6 points) Describe the major advantage of an explicit free list allocator over an implicit list allocator. Also describe the situation in which this advantage is very large. (1-2 sentences)

## Question 9: Memory Bugs

(20 total points)

Consider the following C code where main shows the intended use of the functions relating to **struct** foo.

```
1  struct foo {
2      int x;
3      int y;
4      int* p;
5  };
6
7  struct foo* make_foo() {
8      struct foo* foo = malloc(sizeof(struct foo*));
9
10     foo->x = foo->y * 2;
11     foo->y = 2;
12
13     foo->p = malloc(sizeof(int));
14     *(foo->p) = 42;
15
16     return foo;
17 }
18
19 void update_foo(struct foo* foo) {
20     int num = *(foo->p);
21     foo->p = malloc(sizeof(int));
22     *(foo->p) = num + 1;
23 }
24
25 void free_foo(struct foo* foo) {
26     free(foo);
27     free(foo->p);
28 }
29
30 int main() {
31     struct foo* f = make_foo();
32     for(int i = 0; i < 1000000; i++) {
33         update_foo(f);
34     }
35     free_foo(f);
36 }
```

Questions are on the following page.

(a) (10 points) The function `make_foo` has two memory-related bugs. Find them and briefly describe them (no more than 1 sentence each).

(b) (5 points) The function `update_foo` also has a memory-related bug. Find it briefly describe it (no more than 1 sentence).

(c) (5 points) The function `free_foo` also has a memory-related bug. Find it briefly describe it (no more than 1 sentence).

**Question 10: Extra Credit***(10 total points)*

```

unsigned mystery(unsigned x) {
    x = x - 1;
    x = x | (x >> 1);
    x = x | (x >> 2);
    x = x | (x >> 4);
    x = x | (x >> 8);
    x = x | (x >> 16);
    x = x + 1
    return x;
}

```

- (a) (5 points) Show the step-by-step values of `x` for the execution of `mystery(34)`. Fill in the blanks with value of `x` **after** that line runs. Write **in binary**, and only show the **bottom 8 bits**. Note that `34 == 0x22 == 0b 0010 0010`.

`x = x - 1;` \_\_\_\_\_

`x = x | (x >> 1);` \_\_\_\_\_

`x = x | (x >> 2);` \_\_\_\_\_

`x = x | (x >> 4);` \_\_\_\_\_

`x = x | (x >> 8);` \_\_\_\_\_

`x = x | (x >> 16);` \_\_\_\_\_

`x = x + 1;` \_\_\_\_\_

- (b) (5 points) Describe succinctly what the code above does.

UW NetID: \_\_\_\_\_

This page intentionally left blank.

# CSE 351 Reference Sheet (Final)

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

$2^0$	$2^1$	$2^2$	$2^3$	$2^4$	$2^5$	$2^6$	$2^7$	$2^8$	$2^9$	$2^{10}$
1	2	4	8	16	32	64	128	256	512	1024

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
$10^3$	Kilo-	K	$2^{10}$	Kibi-	Ki
$10^6$	Mega-	M	$2^{20}$	Mebi-	Mi
$10^9$	Giga-	G	$2^{30}$	Gibi-	Gi
$10^{12}$	Tera-	T	$2^{40}$	Tebi-	Ti
$10^{15}$	Peta-	P	$2^{50}$	Pebi-	Pi
$10^{18}$	Exa-	E	$2^{60}$	Exbi-	Ei
$10^{21}$	Zetta-	Z	$2^{70}$	Zebi-	Zi
$10^{24}$	Yotta-	Y	$2^{80}$	Yobi-	Yi

## Sizes

C type	Suffix	Size
char	b	1
short	w	2
int	l	4
long	q	8

## IEEE 754 FLOATING-POINT STANDARD

Value:  $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$

Bit fields:  $(-1)^S \times 1.M \times 2^{(E-\text{bias})}$

where Single Precision Bias = 127,

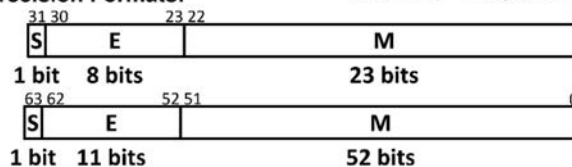
Double Precision Bias = 1023.

## IEEE 754 Symbols

Exponent	Fraction	Object
0	0	$\pm 0$
0	$\neq 0$	$\pm$ Denorm
1 to MAX - 1	anything	$\pm$ Fl. Pt. Num.
MAX	0	$\pm\infty$
MAX	$\neq 0$	NaN

S.P. MAX = 255, D.P. MAX = 2047

## IEEE Single Precision and Double Precision Formats:



## Assembly Instructions

<b>mov a, b</b>	Copy from a to b.
<b>movs a, b</b>	Copy from a to b with sign extension. Needs <i>two</i> width specifiers.
<b>movz a, b</b>	Copy from a to b with zero extension. Needs <i>two</i> width specifiers.
<b>lea a, b</b>	Compute address and store in b. <i>Note:</i> the scaling parameter of memory operands can only be 1, 2, 4, or 8.
<b>push src</b>	Push <i>src</i> onto the stack and decrement stack pointer.
<b>pop dst</b>	Pop from the stack into <i>dst</i> and increment stack pointer.
<b>call &lt;func&gt;</b>	Push return address onto stack and jump to a procedure.
<b>ret</b>	Pop return address and jump there.
<b>add a, b</b>	Add from a to b and store in b (and sets flags).
<b>sub a, b</b>	Subtract a from b (compute $b-a$ ) and store in b (and sets flags).
<b>imul a, b</b>	Multiply a and b and store in b (and sets flags).
<b>and a, b</b>	Bitwise AND of a and b, store in b (and sets flags).
<b>sar a, b</b>	Shift value of b <i>right (arithmetic)</i> by a bits, store in b (and sets flags).
<b>shr a, b</b>	Shift value of b <i>right (logical)</i> by a bits, store in b (and sets flags).
<b>shl a, b</b>	Shift value of b <i>left</i> by a bits, store in b (and sets flags).
<b>cmp a, b</b>	Compare b with a (compute $b-a$ and set condition codes based on result).
<b>test a, b</b>	Bitwise AND of a and b and set condition codes based on result.
<b>jmp &lt;label&gt;</b>	Unconditional jump to address.
<b>j* &lt;label&gt;</b>	Conditional jump based on condition codes ( <i>more on next page</i> ).
<b>set* a</b>	Set byte based on condition codes.

## Conditionals

Instruction	(op) s, d	test a, b	cmp a, b
<b>je</b> "Equal"	d (op) s == 0	b & a == 0	b == a
<b>jne</b> "Not equal"	d (op) s != 0	b & a != 0	b != a
<b>js</b> "Sign" (negative)	d (op) s < 0	b & a < 0	b-a < 0
<b>jns</b> (non-negative)	d (op) s >= 0	b & a >= 0	b-a >= 0
<b>jg</b> "Greater"	d (op) s > 0	b & a > 0	b > a
<b>jge</b> "Greater or equal"	d (op) s >= 0	b & a >= 0	b >= a
<b>jl</b> "Less"	d (op) s < 0	b & a < 0	b < a
<b>jle</b> "Less or equal"	d (op) s <= 0	b & a <= 0	b <= a
<b>ja</b> "Above" (unsigned >)	d (op) s > 0U	b & a < 0U	b > a
<b>jb</b> "Below" (unsigned >)	d (op) s < 0U	b & a > 0U	b < a

## Registers

Name	Convention	Name of "virtual" register		
		Lowest 4 bytes	Lowest 2 bytes	Lowest byte
%rax	Return value – Caller saved	%eax	%ax	%al
%rbx	Callee saved	%ebx	%bx	%bl
%rcx	Argument #4 – Caller saved	%ecx	%cx	%cl
%rdx	Argument #3 – Caller saved	%edx	%dx	%dl
%rsi	Argument #2 – Caller saved	%esi	%si	%sil
%rdi	Argument #1 – Caller saved	%edi	%di	%dil
%rsp	Stack Pointer	%esp	%sp	%spl
%rbp	Callee saved	%ebp	%bp	%bpl
%r8	Argument #5 – Caller saved	%r8d	%r8w	%r8b
%r9	Argument #6 – Caller saved	%r9d	%r9w	%r9b
%r10	Caller saved	%r10d	%r10w	%r10b
%r11	Caller saved	%r11d	%r11w	%r11b
%r12	Callee saved	%r12d	%r12w	%r12b
%r13	Callee saved	%r13d	%r13w	%r13b
%r14	Callee saved	%r14d	%r14w	%r14b
%r15	Callee saved	%r15d	%r15w	%r15b

## C Functions

**void\*** malloc(**size\_t** size):

Allocate size bytes from the heap.

**void\*** calloc(**size\_t** n, **size\_t** size):

Allocate n\*size bytes and initialize to 0.

**void** free(**void\*** ptr):

Free the memory space pointed to by ptr.

**size\_t** sizeof(**type**):

Returns the size of a given type (in bytes).

**char\*** gets(**char\*** s):

Reads a line from stdin into the buffer.

**pid\_t** fork():

Create a new child process (duplicates parent).

**pid\_t** wait(**int\*** status):

Blocks calling process until any child process exits.

**int** execv(**char\*** path, **char\*** argv[]):

Replace current process image with new image.

## Virtual Memory Acronyms

<b>MMU</b>	Memory Management Unit	<b>VPO</b>	Virtual Page Offset	<b>TLBT</b>	TLB Tag
<b>VA</b>	Virtual Address	<b>PPO</b>	Physical Page Offset	<b>TLBI</b>	TLB Index
<b>PA</b>	Physical Address	<b>PT</b>	Page Table	<b>CT</b>	Cache Tag
<b>VPN</b>	Virtual Page Number	<b>PTE</b>	Page Table Entry	<b>CI</b>	Cache Index
<b>PPN</b>	Physical Page Number	<b>PTBR</b>	Page Table Base Register	<b>CO</b>	Cache Offset