University of Washington — Computer Science and Engineering

Winter 2026 - Instructor: Elba Garza - 2026-03-18

CSE 351 FINAL

Last Name:		
First Name:		
Student ID Number:		
Name of person to your Left | Right:		
All work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CSE351 who haven't taken it yet. Violation of these terms could result in a failing grade and a report to CSSC. (please sign)		

Do not turn the page until 14:30.

Instructions

- This exam contains 10 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The last page is a reference sheet. Please detach it from the rest of the exam.
- The exam is closed book (no laptops, tablets, wearable devices). You are allowed two pages (US letter/A4, double-sided) of handwritten notes. Scientific calculators are allowed.
- Please silence and put away all cell phones and other mobile or noise-making devices.
- Remove all hats, headphones, and watches.
- You have 110 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read all questions first and start where you feel the most confident.
- Relax. You are here to learn. You can do it! 🍀

Question M1: Number Representation

A) Suppose we have a variable `x` whose hexadecimal representation is `0xFF 00 00 00` (**Note that any leading zeros are omitted**).

1) Assuming `x` was encoded as one of the following types, which type(s) would make `x` negative? CIRCLE ALL THAT APPLY:

`unsigned int` `int` `float` `long` `unsigned long`

2) Assuming `x` was encoded as one of the following types, which type(s) would make `x` the most positive? If multiple answers are equal, CIRCLE them all:

`unsigned int` `int` `float` `long` `unsigned long`

3) Assuming `x` was encoded as one of the following types, which type(s) would make `x` have the largest magnitude? If multiple answers are equal, CIRCLE them all:

`unsigned int` `int` `float` `long` `unsigned long`

B) Suppose we have:

```
signed int z = [unknown];      signed short s = (signed short) z;
```

If `s` has the hexadecimal representation `0x8000`, which value(s) could replace `[unknown]` so that `z` and `s` have the same decimal value? CIRCLE ALL THAT APPLY:

`0x00008000` `0xFFFF8000` `0x88888000` `0x11118000` `0x10008000`

Question M2: Pointers and Memory

For this problem we are using a 64-bit x86-64 machine (little-endian). A partial view of the current state of memory (values in hex) is shown below:

```
// num is shown in memory
char* msg = "wow!";
int num = 0x74;
char* str = 0xE8;
```

Word Addr	+0	+1	+2	+3	+4	+5	+6	+7
0xC8	82	55	72	AB	CE	0A	B1	00
0xD0	91	A2	B3	6E	74	00	00	00
0xD8	5A	00	88	77	66	55	44	33
0xE0	00	00	00	74	20	30	00	69
0xE8	68	6F	6D	65	00	64	32	55

A) **How many bytes** are allocated by the declarations and initializations in the code above?

bytes

B) What is the result of **`printf`**`("%s", str)`, printing `str` as a string? A partial ASCII chart is provided for reference.

Hex	0x64	0x65	0x66	0x67	0x68	0x69	0x6A	0x6B	0x6C	0x6D	0x6E	0x6F	0x70
Char	d	e	f	g	h	i	j	k	l	m	n	o	p

C) For the following C expressions, compute their **C type** and (hexadecimal) **value** (no leading zeroes). Then, assuming `%rdi` holds `str` and `%rsi` holds `&num`, fill in the blanks for the corresponding x86-64 instructions.

Expression	C Type	Hex Value	X86-64 Instruction
`dl = *(str + 1)`		0x	_____b _____, %dl
`rcx = &num - 2`		0x	_____q _____, %rcx

Question M3: Design Questions

A) Which of these are **benefits** of having function frames (local variables, return address, etc.) on the Stack instead of in blocks in the Heap? SELECT ALL THAT APPLY:

- ☐ Allocating and deallocating memory is faster on the Stack than in the Heap
- ☐ Saving a register value by pushing it on the Stack is more memory efficient than storing it in a new block in the Heap
- ☐ Consecutively called functions have contiguous frames in memory, so functions can access additional arguments in their caller's frame
- ☐ LIFO structure allows for easy access to the return address when exiting a function
- ☐ Functions can allocate data that persist after they return without leaking memory

B) Describe **one benefit** of using the Heap for user input instead of in the Stack:

Describe (1-2 sentences):

C) Without changing anything else about x86 register conventions, could we change `%rax` to be *callee-saved* instead of *caller-saved*? Explain why or why not.

Can change to callee-saved? (CIRCLE ONE):	Yes	No
Explain (1-3 sentences):		

Question M4: C & Assembly

Consider the following x86-64 assembly for a function `mystery`:

```
mystery:
        pushq   %rbx                # Line 1
        movq    %rdi, %rax          # Line 2
        movq    %rax, %rbx          # Line 3
        jmp     .L2                 # Line 4
.L3:
        movb    (%rax), %dl         # Line 5
        xor     $0x20, %dl          # Line 6
        movb    %dl, (%rax)         # Line 7
        addq    $1, %rax            # Line 8
.L2:
        movzbl  (%rax), %edx        # Line 9
        testb   %dl, %dl            # Line 10
        jne     .L3                 # Line 11
        movq    %rbx, %rax          # Line 12
        popq    %rbx                # Line 13
        ret                         # Line 14
```

A) Fill in the missing C code that is equivalent to the x86-64 assembly above:

```
_____________ mystery ( _____________ param) {

    _____________ var = _____________ ;

    while ( _____________ ) {

        _____________ ^= 0x20;

        var += 1;

    }

    return _____________;

}
```

B) If we called `mystery` with `"Abacadaba"` as a parameter, what would be returned from the function call? Answer in the format "_____"

Hint: *in ASCII, 'a' is 0x61, 'b' is 0x62, 'c' 0x63, … , 'A' is 0x41, 'B' 0x42, 'C' 0x43…*

For the following instructions, write an equivalent line(s) of assembly using the replacement instruction(s) suggested.

C)
```
popq %rbx

        movq  _________ , _________

        addq  _________ , _________
```

D)
```
retq

        movq  _________ , %r10

        addq  _________ , _________

        jmp   *_________ // hint: a register may go in this blank \_(0o0)_/
```

Question M5: Procedures & The Stack

Consider the following C code, which defines a struct for a linked list node and finds the sum of a linked list of nodes:

```
// struct definition
struct ll_node {
  int data;
  struct ll_node* next;
};

// find the sum of a linked list
int sum_ll(struct ll_node* start) {
  if (start->next == 0) {
    return start->data;
  }
  return start->data +
         sum_ll(start->next);
}
```

```
// create a linked list and sum it
int main() {
  // 0 -> 1 -> 2 -> 3 -> 4
  struct ll_node four = {4, 0};
  struct ll_node three = {3, &four};
  struct ll_node two = {2, &three};
  struct ll_node one = {1, &two};
  struct ll_node root = {0, &one};

  int sum = sum_ll(&root);
  printf("%d\n", sum);
}
```

Also consider the disassembly for `sum_ll`:

```
#:  0000000000401126 <sum_ll>:
1     401126:  48 8b 47 08           mov    0x8(%rdi),%rax
2     40112a:  48 85 c0              test   %rax,%rax
3     40112d:  74 0f                 je     40113e <sum_ll+0x18>
4     40112f:  53                    push   %rbx
5     401130:  8b 1f                 mov    (%rdi),%ebx
6     401132:  48 89 c7              mov    %rax,%rdi
7     401135:  e8 ec ff ff ff        call   401126 <sum_ll>
8     40113a:  01 d8                 add    %ebx,%eax
9     40113c:  5b                    pop    %rbx
10    40113d:  c3                    ret
11    40113e:  8b 07                 mov    (%rdi),%eax
12    401140:  c3                    ret
```

A) What is the return address to `sum_ll` that is saved on the stack for the recursive calls of `sum_ll`?

B) How many bytes does `sum_ll` take up in its executable?

bytes

C) How many **total stack frames** are created by the above program?

frames

D) Fill out the stack snapshot below at the point where the stack is largest. The number of boxes is arbitrary. Use `???` for unknown words.

	Value	Description
`...`	`<...>`	`<main`'s stack frame>
`0x7fffffffdff8`	`???`	<ret addr to main>
`0x7fffffffdff0`		
`0x7fffffffdfe8`		
`0x7fffffffdfe0`		
`0x7fffffffdfd8`		
`0x7fffffffdfd0`		
`0x7fffffffdfc8`		
`0x7fffffffdfc0`		
`0x7fffffffdfb8`		
`0x7fffffffdfb0`		
`0x7fffffffdfa8`		
`0x7fffffffdfa0`		

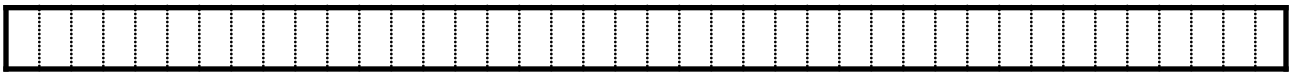
Question F6: Structs

Consider the following definition for a struct `pokemon`:

```
typedef struct {
  short* evolution;
  char name[3];
  int level;
  long hp;
  char type;
} pokemon;
```

A) Use the box below to draw out the bytes in an instance of the `pokemon` struct. Clearly **label each field and its size (in bytes)**. Clearly **specify any internal and external fragmentation**. **Draw a clear line indicating the end of the struct**; some boxes may be unnecessary. Then, state the amount of internal and external fragmentation, as well as the total size of the struct (including any fragmentation).

`pokemon` (1 box = 1 byte):



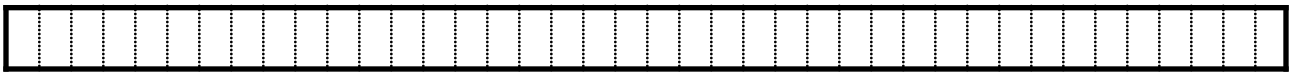
Internal Fragmentation: _____ bytes

External Fragmentation: _____ bytes

Total Size: _____ bytes

B) Can we reorder the fields in `pokemon` to reduce the size of an instance? If so, draw out the diagram using the same rules as above. If not, provide a justification of why. You only need to draw out the diagram or provide a justification, **not both**.

Smaller `pokemon` struct layout (**if possible**) (1 box = 1 byte):



Justification (**if not possible**) (1-2 sentences):

C) Suppose you have an array that contains 3 instances of `pokemon` defined as follows:

```
pokemon* pokemons[] = {pokemon1, pokemon2, pokemon3};
```

What is an expression of C code that would set the `evolution` field of `pokemon3` to be the same as the `evolution` field of `pokemon1`?

Your expression **<u>CANNOT contain</u>** the variables `pokemon1`, `pokemon2`, or `pokemon3`.

C Expression: _____

Question F7: Caching

A **next-line prefetcher** is a component that **fetches the next cache block** early. **On a cache miss**, the cache will pull in a requested block from memory. At the same time, the next-line prefetcher will **also pull in the block that immediately follows in memory**.

Assume a cache with a next-line prefetcher starts cold and has the following parameters:

- Associativity: 1
- Block size: 8 B
- Capacity: 512 B
- Write-back, write-allocate
- LRU replacement policy

Consider the following C code snippets:

```
// Snippet 1
int sum = 0;
int arr[1024];   // &arr[0] = 0x10
for (int i = 0; i < 1024; i++) {
    sum += arr[i];
}
```

```
// Snippet 2
#define LEAP = 4
int sum = 0;
int arr[1024];   // &arr[0] = 0x10
for (int i = 0; i < 1024; i += LEAP) {
    sum += arr[i];
}
```

A) Assuming `sum` and `i` are stored in registers, calculate the miss rate of each snippet on the above cache as a percentage:

Snippet 1: _____ % Snippet 2: _____ %

B) Compared to an identical cache with no prefetching, does the presence of the next-line prefetcher improve the performance of Snippet 1? Why or why not?

Performance (CIRCLE ONE):	Improves	Does NOT improve
Explain (1-3 sentences):		

C) Compared to an identical cache with no prefetching, does the presence of the next-line prefetcher improve the performance of Snippet 2? Why or why not?

Performance (CIRCLE ONE):	Improves	Does NOT improve
Explain (2-3 sentences):		

D) Which cache set does the block associated with `arr[7]` map to?

set

E) Which of the following array elements does **NOT** map to the same set as `arr[7]`? (Circle one):

`arr[6]` `arr[134]` `arr[136]` `arr[262]`

F) For each of the following proposed changes, CIRCLE ONE option that describes its effect on the **miss rate** of the code snippets above. Assess each change **independently**: assume all other factors remain the same as they were in the original cache.

1) Change the associativity from 1-way to 2-way:	Increase	No Change	Decrease
2) Change `LEAP` in Snippet 2 from 4 to 3:	Increase	No Change	Decrease
3) Change the cache size from 512 B to 1 KiB:	Increase	No Change	Decrease
4) Change the cache replacement policy to Least **frequently** used:	Increase	No Change	Decrease

Question F8: Memory Allocation

A) Consider the following C code snippet:

```
#:
1   #include <stdlib.h>
2
3   int NUM_STUDENTS = 30;
4   void avg_exam_score(float* midterm, float* final) {
4       char* course = "CSE351";
5       float* grades = (float*) malloc(NUM_STUDENTS * sizeof(float));
6       float* start = grades;
7
8       for (int i = 0; i < NUM_STUDENTS; i++) {
9           *grades = (midterm[i] + final[i]) / 2;
11          grades++;
12      }
13      free(grades);
14  }
```

1) This snippet has a memory-related bug. Identify the line number of the bug and write one line of C code to replace the buggy line that fixes the bug.

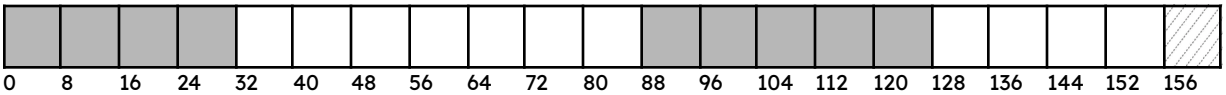
Line number:
Fix (1 line of C code):

2) For the groups of expressions below, CIRCLE the one in each group whose returned **value** is **largest/highest**. Assume the malloc call succeeds and all variables are stored in memory (not registers).

Group 1:	`&NUM_STUDENTS`	`*course`	`start`
Group 2:	`course`	`&start`	`NUM_STUDENTS`
Group 3:	`&course`	`grades`	`&avg_exam_score`

B) For the following questions, consider a dynamic allocator on a 64-bit machine with 8-byte alignment, 8-byte boundary tags, and an explicit free list. Allocated blocks (shaded) have only a header, and free blocks have a header and a footer.

Answer the following questions given the current state of the heap below. Each box represents one word; the final box (address 156) is the end-of-heap block.



1) What is the largest payload that can be allocated without requesting more space on the heap?

malloc ( )

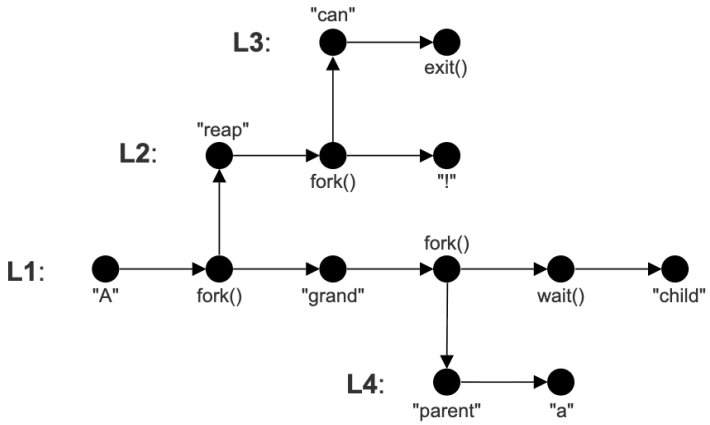
2) Assuming a best-fit allocation strategy, what pointer would be returned by a call to `malloc(2 * sizeof(int))`? Format your answer in **decimal**.

Question F9: Processes

A) Fill in the blanks in the C code to match the behavior of the process diagram. The labels `L1`-`L4` number the four processes created.

```
printf("A");
if (fork()) {
    printf("_____________");
    if (fork()) {
        wait();
        printf("_____________");
    } else {
        printf("_____________");

        printf("_____________");
    }
} else {
    printf("_____________");
    if (fork()) {
        printf("_____________");
    } else {
        printf("_____________");
        exit(0);
    }
}
```



B) For each of the following outputs, CIRCLE whether the output is possible or not possible. If an output is not possible, **CIRCLE the first invalid word** in the output.

`"A grand reap ! can parent a child"`	Possible	Not possible
`"A reap grand a parent can ! child"`	Possible	Not possible
`"child A can ! reap grand parent a"`	Possible	Not possible
`"A grand parent can reap a child !"`	Possible	Not possible

⚠️ *Did you remember to **CIRCLE the first invalid word** on all impossible outputs?* ⚠️

C) Which process is L3's parent, if any? CIRCLE ONE option:

L1 L2 L3 L4 None

D) Which process is L3's grandparent, if any? CIRCLE ONE option:

L1 L2 L3 L4 None

E) When process L3 terminates, who will reap it? CIRCLE ONE option:

`init/systemd` L3's parent L3's grandparent Elba

Question F10: Virtual Memory

A) Circle ONE option to indicate whether these events occur always, sometimes, or never in a single address translation:

1) TLB hit → segmentation fault	Always	Sometimes	Never
2) Page table hit → page fault	Always	Sometimes	Never
3) TLB miss → check page table	Always	Sometimes	Never
4) TLB hit → cache hit	Always	Sometimes	Never

B) Circle ONE option to indicate whether the following statements about virtual memory are true or false.

1) Main memory acts as a "cache" for swap space on disk.	True	False
2) As part of a context switch, we must invalidate the TLB.	True	False
3) Every virtual page is mapped to a location either in physical memory or on disk.	True	False
4) The widths of the virtual page offset and the physical page offset fields are always the same.	True	False
5) Multiple processes can share a single page table.	True	False

C) For a system with the following properties:

- 32-bit virtual addresses,
- 20-bit physical addresses,
- 4 KiB virtual pages,
- An 8-entry, 4-way associative TLB,

Calculate the following:

_____ TLB Index bits _____ TLB Tag bits _____ Page table entries

This page is purposely left blank and can be used for scratch work.

CSE 351 Final Reference Sheet

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

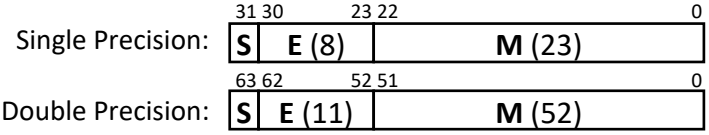
Bit fields: $(-1)^{S} \times 1.M \times 2^{(E-\text{bias})}$

single precision bias = 127

double precision bias = 1023

IEEE 754 Encodings

E	M	Meaning
all zeros	all zeros	± 0
all zeros	non-zero	± denorm num
1 to MAX-1	anything	± norm num
all ones	all zeros	± ∞
all ones	non-zero	NaN



Single Precision:

Double Precision:

Assembly Instructions

mov a, b	Copy from a to b.
movs a, b	Copy from a to b with sign extension. Needs *two* width specifiers.
movz a, b	Copy from a to b with zero extension. Needs *two* width specifiers.
lea a, b	Compute effective address and store in b. *Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8.
push src	Push `src` onto the stack and decrement stack pointer.
pop dst	Pop from the stack into `dst` and increment stack pointer.
call <func>	Push return address onto stack and jump to a procedure.
ret	Pop return address and jump there.
add a, b	Add from a to b and store in b (and sets flags).
sub a, b	Subtract a from b (compute b-a) and store in b (and sets flags).
imul a, b	Multiply a and b and store in b (and sets flags).
and a, b	Bitwise AND of a and b, store in b (and sets flags).
sar a, b	Shift value of b *right* (*arithmetic*) by a bits, store in b (and sets flags).
shr a, b	Shift value of b *right* (*logical*) by a bits, store in b (and sets flags).
shl a, b	Shift value of b *left* by a bits, store in b (and sets flags). Same as **sal**.
cmp a, b	Compare b with a (compute b-a and set condition codes based on result).
test a, b	Bitwise AND of a and b and set condition codes based on result.
jmp <label>	Unconditional jump to address.
j* <label>	Conditional jump based on condition codes (*more on next page*).
set* a	Set byte a to 0 or 1 based on condition codes.

Conditionals

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

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