University of Washington – Computer Science & Engineering

Winter 2018 Instructor: Mark Wyse March 14, 2018

CSE 351 Final Exam

Last Name:	**Solutions**
First Name:	
UW Student ID Number:	
UW NetID (username):	
Academic Integrity Statement: All work on this exam is my own. I had no prior knowledge of the exam contents, nor will I share the contents with others in CSE 351 who haven't taken it yet. Violation of these terms may result in a failing grade. (**please sign**)	

Do not turn the page until 2:30 pm.

Instructions

- This exam contains **16** pages, including this cover page, and 2 reference pages.
- Show scratch work for partial credit but answer in the blanks and boxes provided.
- The last page is a reference sheet. Please detach it from the rest of the exam.
- This exam is **closed book and closed notes** (no laptops, tablets, smartphones, wearable devices, or calculators).
- Please silence/turn-off all cell phones, mobile devices, or other noise-making devices. Remove all hats, headphones, and watches.
- You have 110 minutes to complete this exam.
- Illegible handwriting and/or answers will receive a score of 0.

Advice

- Read each question carefully.
- Read *all* questions first and start where you feel most confident.
- Relax and breathe; you are here to learn.

Question	**1**	**2**	**3**	**4**	**5**	**6**	**Total**
Points Possible	25	6	41	10	9	24	**115**
Points Earned	25	6	41	10	9	24	**115**

Question 1: Warm-up [25 pts.]

True or False [10 pts.]

Answer the following questions by circling True or False. No explanation is needed.

(1) The ISA specifies the names of all registers but does not specify the size of registers.

True / False

(2) Two's Complement is the only valid signed integer representation that could be used when implementing a processor.

True / False

(3) Assume procedure P calls procedure Q and P stores a value in register `%rbp` prior to calling Q. After Q returns control to P, P can safely use the register `%rbp`.

True / False

(4) When programming in C, the *compiler* determines whether local variables are allocated on the stack or stored in registers.

True / False

(5) Your friend is designing a new processor that implements the x86-64 ISA for a new startup. Unfortunately, they aren't a very good at their job, and their processor has a cycle time (clock cycle) of 24 hours. That is, it takes 24 hours to execute a single instruction (assume that all instructions take only a single cycle to execute). *Your friend's slow processor may still be a valid implementation of x86-64.*

True / False

(6) In C, it is safe (there is no possibility of losing data) when casting from type `int` (4-byte integer) to type `float` (4-byte IEEE floating point).

True / False

(7) In C, it is safe (there is no possibility of losing data) when casting from type `int` (4-byte integer) to type `double` (8-byte IEEE floating point).

True / False

(8) In Java, all non-primitive variables are references to objects.

True / False

(9) In the context of memory allocators, and compared to an implicit free list, an allocator using an explicit free list is much faster at allocating blocks when most of the memory is free (unused).

True / False -- faster when memory is full (explicit free list size shrinks as memory is allocated)

(10) You learned at least one new thing in 351 this quarter.

True / False

Fill in the Blank / Short-Answer [15 pts.]

Answer the following questions in the spaces provided. Illegible answers will not be graded.

(11) A Process provides two key abstractions to each program. What are they? [2 pts.]

Logical Control Flow	Private Address Space

(12) What are the two types of *locality*? Provide the *name* and *definition* (one English sentence/fragment or less) for each. [4 pts.]

	Name	**Definition**
1.	Temporal	Recently referenced items are likely to be referenced again in the near future.
2.	Spatial	Items/data with nearby addresses tend to be referenced close together in time.

(13) Name the type of synchronous exception for each of the following definitions: [3 pts.]

a. An unintentional, but possibly recoverable exception caused by the execution of an instruction.

Fault

b. An unintentional and unrecoverable exception, caused for example by a machine check, that stops the execution of the current program.

Abort

c. An intentional transfer of control to the Operating System (Kernel) to perform some function as requested by the user program.

Trap

(14) In IEEE 754 single precision floating point (32-bit), 8 bits are used to represent the exponent. Assume we have a new floating-point type that uses 4 bits for the exponent, but otherwise follows all the same conventions as IEEE 754 floating point. What is the **bias** for this new type? [1 pt.]

7, formula is bias = (2^(k-1) – 1)

(15) What are the three types of *Cache Misses*? [3 pts.]

Compulsory	Capacity	Conflict

(16) Assume we are executing code on a machine that uses ***k***-bit virtual addresses, and each addressable memory location stores ***b***-bytes. *What is the total size of the addressable virtual memory space on this machine?* [2 pts.]

$\boxed{(2^k) * b}$

Question 2: Number Representation [6 pts.]

In computer graphics, a pixel's color is determined by a combination of the colors red (R), green (G), and blue (B). Let's assume that we have a new display where each pixel will display either red, greed, or blue. A pixel will only display a single color at a time, and it must be one of R, G, or B, so we can use pixels to encode values in base 3! Assume we use the encoding 0↔R, 1↔G, 2↔B. For example, $6 = 2*(3^1) + 0*(3^0)$ would be encoded as BR.

(A) What is the *unsigned* decimal value of the set of pixels displaying **GGRB**? [2 pts.]

$\boxed{38}$

1*3^3 + 1*3^2 + 0*3^1 + 2*3^0 = 27 + 9 + 0 + 2 = 38

(B) If we have **7 bits** of *binary* data that we want to store as a set of pixels, how many *pixels* would it take to store that same data. [2 pts.]

$\boxed{\text{5 pixels}}$

7 bits can represent 128 things. Powers of 3: 1, 3, 9, 27, 81, <u>243</u>. So, we need 5 pixels, which can represent up to 242 things.

(C) Assume we can perform the *left shift* operation on a set of pixels. Similar to in binary, we will shift in the 0 value, or equivalently a Red pixel. For example, our pixels from part (A) left-shifted by 1 would become **GGRBR**, or `GGRB << 1 = GGRBR`. What arithmetic operation occurs when left shifting by 1? [2 pts.]

$\boxed{\text{multiply by 3}}$

1*3^4 + 1*3^3 + 0*3^2 + 2*3^1 + 0*3^0 = 81 + 27 + 0 + 2*3 + 0 = 114 = 3*38

Question 3: Caching, Address Translation, and Virtual Memory [41 pts.]

This question is a multi-part question that deals with several memory related topics we have studied. Each part is designed to be independent from the others.

Part I. Caching [14 pts.]

Assume we are using a computer system with a physically addressed data cache. Physical addresses are 7 bits in length. The data cache has a total capacity of 64 bytes, is 2-way set associative, and the cache block size is 4 bytes. The cache uses LRU replacement, write-back, and write-allocate policies. Assume the state of the cache is as shown below (dirty bit omitted due to space). Assume our system is **Little Endian** and stores multi-byte data in little endian form (like x86-64 systems).

Data Cache

Index	Tag	Valid	B0	B1	B2	B3	Tag	Valid	B0	B1	B2	B3
0	-	0	-	-	-	-	0	1	FB	17	47	E6
1	1	1	D9	C6	07	01	2	1	D9	C6	07	01
2	1	1	FA	34	8C	14	-	0	-	-	-	-
3	1	1	8D	76	26	5F	0	1	1B	BB	CB	34
4	2	0	2F	F1	6A	E8	3	1	2F	F1	6A	E8
5	-	0	-	-	-	-	3	1	76	C6	B2	D3
6	-	0	-	-	-	-	-	0	-	-	-	-
7	3	0	F5	CA	08	16	2	1	35	89	85	4B

(A) How many bits are used for the cache index, offset, and tag fields? [3 pts.]

Tag bits	Index bits	Offset bits
2	3	2

(B) How many management bits (bits *other* than the block data) are there in every line in the data cache? [1 pt.]

Management bits
4 (Tag bits + valid + dirty)

(C) For each of the following cache accesses, *provide the cache tag, index, and offset (in hex)*. Then, *determine if the request results in a cache hit* (mark "Y" or "N"). If a cache hit occurs, *provide the data returned from the cache* **for the access size requested (given in bytes)** *in hex*. [10 pts.]

Physical Address	Request Size (Bytes)	Tag	Index	Offset	Hit (Y/N)	Data Returned
0x24	**2**	0x1	0x1	0x0	Y	0xC6D9
0x51	**3**	0x2	0x4	0x1	N	n/a

0x51 = 0b 10 100 01, 0x24 = 0b 01 001 00

Part II. Address Translation [22 pts.]

As in Part I, assume our computer system has 7-bit physical addresses and is a little-endian system. Assume virtual addresses are 8-bits, virtual and physical pages of memory are 16 bytes in size, and the TLB holds 4 entries and is 2-way set associative. Assume the state of the TLB and Page Table are as shown below. The tag and PPN in the TLB are shown in hex. The PTE and PPN in the Page Table are shown in hex.

TLB

Set	Tag	Valid	PPN	Tag	Valid	PPN
0	3	1	7	1	0	1
1	-	0	-	7	1	2

Page Table

PTE	PPN	Valid	PTE	PPN	Valid
0	2	0	**8**	4	0
1	-	0	**9**	6	1
2	4	1	**A**	-	0
3	5	1	**B**	-	0
4	5	0	**C**	1	1
5	7	1	**D**	-	0
6	1	0	**E**	0	1
7	-	0	**F**	2	1

(A) How many bits are used for the VPN, VPO, TLB tag, TLB index, PPN, and PPO? [6 pts.]

VPN	VPO	TLB Tag	TLB Index	PPN	PPO
4 bits	4 bits (16B pages)	3 bits	1 bit	3 bits	4 bits (16B pages)

(B) For each of the following virtual addresses, *provide the VPN, VPO, TLBT, and TLBI (in hex)*. Then, *determine if a TLB Hit occurs and if a Page Fault occurs* (mark "Y" or "N" for each). If possible, *provide the PPN and compute the physical address (in hex)*. If the PPN cannot be determined, leave the PPN and PA entries blank. [16 pts.]

VA	VPN	VPO	TLB Tag	TLB Index	TLB Hit (Y/N)	Page Fault (Y/N)	PPN	PA
0xF4	F	4	7	1	Y	N	2	24
0x31	3	1	1	1	N	N	5	51

0xF4 = 0b 1111 0100, 0x31 = 0b 0011 0001

Part III. Cache and TLB Performance [5 pts.]

Assume we execute the C code given below on the computer system we have described in Parts I and II. However, assume that both the Cache and TLB start **cold** (empty). Assume that the array of structs named `structArr` is allocated at virtual address 16, or 0x10, and virtual pages are mapped to physical pages in a linear manner, starting with physical page 0 (PPN = 0).

```
typedef struct {
  byte b;
  short s;
} myStruct;

#define N 16

int main(int argc, char **argv) {
  // Assume structArr is located at address 16 (0x10)
  myStruct *structArr = (myStruct*) malloc(N * sizeof(myStruct));
  // code to initialize array elements omitted
  short sum = 0;
  for (int i = 0; i < N; i++) {
    sum += structArr[i].s;
  }
}
```

(A) Compute *both* the Cache and TLB miss rates for the accesses to `structArr` in the code above. Remember, the cache and TLB start cold. Express your answers as a percentage. [2 pts.]

Cache Miss Rate	TLB Miss Rate
100% (one struct per cache line)	25% (a page holds 4 structs)

(B) What would happen to the TLB miss rate if the page size was doubled? *Circle your answer.* [1 pt.]

a. Increase
[✓ b. Decrease]
c. No Change

(C) What would happen to the Cache miss rate if the capacity of the cache was doubled while keeping the associativity and block size fixed? *Circle your answer.* [1 pt.]

a. Increase
b. Decrease
[✓ c. No Change]

(D) This question is independent from the previous questions in Part III. *Compute the Average Memory Access Time (AMAT)* assuming it takes 100 ns to get a block of data from main memory, the data cache has a hit time of 2 ns, and the data cache miss rate is 2%. Don't forget to use the correct units for your answer! [1 pt.]

4 ns
AMAT = HT + MR*MP =
2 + 0.02*100

Question 4. Pointers and Memory [10 pts.]

For this problem we are using a 64-bit x86-64 machine (**little endian**). Below is the recursive function `rfun` from the midterm exam, *showing where the code is stored in memory*.

```
00000000004005e6 <rfun>:
  4005e6: 0f b6 07              movzbl (%rdi),%eax
  4005e9: 84 c0                 test   %al,%al
  4005eb: 74 13                 je     400600 <rfun+0x1a>
  4005ed: 53                    push   %rbx
  4005ee: 48 0f be d8           movsbq %al,%rbx
  4005f2: 48 83 c7 01           add    $0x1,%rdi
  4005f6: e8 eb ff ff ff        callq  4005e6 <rfun>
  4005fb: 48 01 d8              add    %rbx,%rax
  4005fe: eb 06                 jmp    400606 <rfun+0x20>
  400600: b8 00 00 00 00        mov    $0x0,%eax
  400605: c3                    retq
  400606: 5b                    pop    %rbx
  400607: c3                    retq
```

(A) What are the values (in hex) stored in each register shown after the following x86-64 instructions are executed? *Remember to use the appropriate bit widths.* [6 pts.]

```
movb    (%rax), %cl
leaq    8(%rax, %rsi, 2), %rcx
movswl  (%rax, %rsi), $r8d
```

Register	Value (in hex)
`%rax`	0x 0000 0000 0040 05e6
`%rsi`	0x 0000 0000 0000 0005
`%cl`	0x0f
`%rcx`	0x 0000 0000 0040 05f8
`%r8d`	0x 0000 1374

(B) Complete the C code below to fulfill the behaviors described in the inline comments using pointer arithmetic. Let **char* cp = 0x4005ee**. [4 pts.]

```
char v1 = *(cp + __2___);                         // set v1 = 0xbe
int* v2 = (int*)((____short____*)cp - 8);         // set v2 = 0x4005de
```

The only 0xbe byte in rfun is found at address 0x4005f0, 2 bytes beyond cp.

The difference between v2 and cp is 16 bytes. Since by pointer arithmetic we are moving 8 “things” away, cp must be cast to a pointer to a data type of size 2 bytes, such as short.

Question 5: Processes [9 pts.]

(A) The block of code shows an infinite loop of the fork() process being called (a.k.a. a forkbomb), briefly explain why this is bad, specifically how it affects the system that it is being run on. [2 pts.]

```
void forkbomb() {
  while (1) {
    fork();
  }
}
```

Answer: **A forkbomb will starve the system of its resources by creating an infinite number of processes. The system can support a finite number of processes, and will eventually run out of process IDs to assign.**

(B) How many total processes are created from the following block of code? Assume the original process is counted in the total number of processes. [2 pts.]

```
for (int i = 0; i < n; i++) {
  fork();
}
```

Answer: There will be 2^n processes created. If we unroll the loop this code is equivalent to n fork() calls in sequence. The original process will call fork() n times, the second process will call fork() n-1 times, etc. This results in a doubling of processes at each fork, thus starting with the original process, and doubling n times, gives 2^n total processes.

(C) List three possible outputs of the following block of code. Write your answers into the right hand column of the table below. Note: there are more than three possible outputs; any three will suffice. [3 pts.]

```
int main() {
   int x = 1;
   printf("%d ", x);
   if (fork() != 0) {
      x = x << 2;
      printf("%d ", x);
      fork();
      x = x << 1
      printf("%d ", x);
   } else {
      x = x << 4
      printf("%d ", x);
   }
   exit(0);
}
```

Write your three answers in this box:

The original process prints 1 – 4 – 8 and the second child process prints 8 either before or after the original process prints 8. So those two processes always produce the output sequence 1 – 4 – 8 – 8. The first child process prints 16. That can happen any time after the first process prints 1. So the possible output sequences are
1 16 4 8 8
1 4 16 8 8
1 4 8 16 8
1 4 8 8 16

(D) In the following blanks, write "Y" for yes or "N" for no if the following need to be updated when `execv` is run on a process. [2 pts.]

Page Table ____**Y**____   PTBR ___**N**_____   Stack ___**Y**_____   Code ___**Y**_____

The process already has a page table, so the PTBR does not need to be updated, but the old PTEs need to be invalidated. We replace/update the old process image's virtual address space, including Stack and Code.

Question 6: Procedures & The Stack [24 pts.]

Consider the following x86-64 assembly and C code for the recursive function `rfun`.

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (long)*s;
    s++;
    return temp + rfun(s);
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "Yay!";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

```
00000000004005e6 <rfun>:
  4005e6: 0f b6 07              movzbl (%rdi),%eax
  4005e9: 84 c0                 test   %al,%al
  4005eb: 74 13                 je     400600 <rfun+0x1a>
  4005ed: 53                    push   %rbx
  4005ee: 48 0f be d8           movsbq %al,%rbx
  4005f2: 48 83 c7 01           add    $0x1,%rdi
  4005f6: e8 eb ff ff ff        callq  4005e6 <rfun>
  4005fb: 48 01 d8              add    %rbx,%rax
  4005fe: eb 06                 jmp    400606 <rfun+0x20>
  400600: b8 00 00 00 00        mov    $0x0,%eax
  400605: c3                    retq
  400606: 5b                    pop    %rbx
  400607: c3                    retq
```

(A) How much space (in bytes) does this function take up in our final executable? [2 pts.]

34 bytes

(B) The compiler automatically creates labels it needs in assembly code. How many labels are used in `rfun` (including the procedure itself)? [2 pts.]

3

(C) In terms of the C function, what value is being saved on the stack? [2 pts.]

***s**

(D) What is the return address to `rfun` that gets stored on the stack during the recursive calls (in hex)? [2 pts.]

`0x4005fb`

(E) Assume main calls `rfun` with `char *s = "Yay!"` and then prints the result using the `printf` function, as shown in the C code above. Assume `printf` does not call any other procedure. Starting with (and including) `main`, how many total stack frames are created, and what is the maximum depth of the stack? [2 pts.]

Total Frames: **7**	Max Depth: **6**

```
main -> rfun(s) -> rfun(s+1) -> rfun(s+2) -> rfun(s+3) -> rfun(s+4)
     -> printf()
```

`The recursive call to rfun(s+4), which handles the null-terminator in the string does create a stack frame since we consider the return address pushed to the stack during a procedure call to be part of the callee’s stack frame.`

(F) Assume main calls `rfun` with `char *s = "Yay!"`, as shown in the C code. After `main` calls `rfun`, we find that the return address to `main` is stored on the stack at address `0x7fffffffdb38`. On the first call to `rfun`, the register `%rdi` holds the address `0x4006d0`, which is the address of the input string "Yay!" (i.e. `char *s == 0x4006d0`). Assume we stop execution prior to executing the `movsbq` instruction (address `0x4005ee`) during the **fourth** call to `rfun`. [14 pts.]

*For each address in the stack diagram below, fill in both the **value** and a **description** of the entry.*

*The **value** field should be a hex value, an expression involving the C code listed above (e.g., a variable name such as `s` or `r`, or an expression involving one of these), a literal value (integer constant, a string, a character, etc.), "unknown" if the value cannot be determined, or "unused" if the location is unused.*

*The **description** field should be one of the following: "Return address", "Saved %reg" (where reg is the name of a register), a short and descriptive comment, "unused" if the location is unused, or "unknown" if the value is unknown.*

Memory Address	Value	Description
0x7fffffffdb48	unknown	%rsp when main is entered
0x7fffffffdb38	0x400616	Return address to main
0x7fffffffdb30	unknown	original %rbx
0x7fffffffdb28	0x4005fb	Return address
0x7fffffffdb20	*s, "Y"	Saved %rbx
0x7fffffffdb18	0x4005fb	Return address
0x7fffffffdb10	*s, *(s+1), "a"	Saved %rbx
0x7fffffffdb08	0x4005fb	Return address
0x7fffffffdb00	*s, *(s+2), "y"	Saved %rbx

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

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

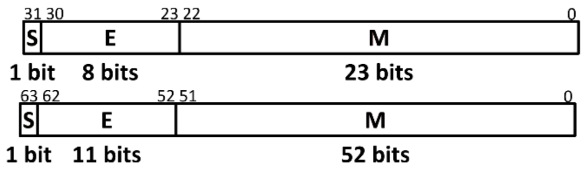
where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

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

IEEE Single Precision and Double Precision Formats:



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 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).
`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 based on condition codes.

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

Registers

		Name of "virtual" register		
Name	**Convention**	**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