Practice Midterm Examination

Disclaimer: *this practice exam is my attempt to help you learn the material in this course. Things in here might not be in the real exam and vice versa. Don't use this as your main study. There might be some typos and/or mistakes.*

1. Implement a function `sumOdd(n)` in the x86-64 assembly language. This function computes and returns the sum of all odd numbers between 1 and n. For example, when $n = 6$, the program computes $1 + 3 + 5 = 9$ and returns 9 as the output.

sumOdd.s

```
        .global sumOdd
        .text
sumOdd:
        xor     %rax, %rax
        mov     $1, %rcx
loop:   cmp     %rdi, %rcx
        jg      done
        add     %rcx, %rax
        add     $2, %rcx
        jmp     loop
done:   ret
```

testsumOdd.c

```
#include<stdio.h>

long sumOdd(long n);

int main() {
  printf("sumOdd(%d) = %ld\n", 5, sumOdd(5));
  printf("sumOdd(%d) = %ld\n", 6, sumOdd(6));
  return 0;
}
```

2. Implement a function `manhattan(A, B, n)` in the x86-64 assembly language. This function computes the Manhattan distance between two vectors $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_n]$ of size n, where

$$manhattan(A, B) = |a_1 - b_1| + |a_2 - b_2| + \cdots + |a_n - b_n|$$

manhattan.s

```
        .global manhattan
        .text
manhattan:
        xor     %rcx, %rcx
        xor     %rax, %rax
loop:   cmp     %rdx, %rcx
        jge     done
        mov     (%rdi), %r8
        sub     (%rsi), %r8
        test    %r8, %r8
        jge     g0
        neg     %r8
g0:     add     %r8, %rax
        inc     %rcx
        add     $8, %rdi
        add     $8, %rsi
        jmp     loop
done:   ret
```

testmanhattan.c

```
#include<stdio.h>

long manhattan(long A[], long B[], long n);

int main() {
  long X[] = {1,2,3,4};
  long Y[] = {2,2,2,2};
  long Z[] = {0,0,0,0};
  printf("Output = %ld\n", manhattan(X, Y, 4));
  printf("Output = %ld\n", manhattan(X, Z, 4));
  printf("Output = %ld\n", manhattan(Y, Z, 4));
  return 0;
}
```

3. Convert the following assembly function into C, Java or Python.

func.s

```
        .global func
        .text
func:   xor     %rax, %rax
loop:   cmp     %rsi, %rax
        jge     done
        mov     (%rdi), %r8
        cmp     %r8, %rdx
        jle     l1
        mov     %r8, (%rcx)
        add     $8, %rcx
l1:     add     $8, %rdi
        inc     %rax
        jmp     loop
done:   mov     $-999, %r8
        mov     %r8, (%rcx)
        ret
```

The function is used with the following C program.

testfunc.c

```
#include<stdio.h>

void func(long A[], long n, long x, long B[]);

int main() {
  long X[] = {1, 2, 3, 4, 5, 6, 7, 8};
  long Y[10];
  long i;

  func(X, 8, 4, Y);
  for(i=0; Y[i]!=-999; i++)
    printf("%ld\n", Y[i]);
  return 0;
}
```

func.c

```
void func(long A[], long n, long x, long B[]) {
  int i, j=0;
  for(i=0; i<n; i++) {
    if (x > A[i]) {
      B[j] = A[i];
      j++;
    }
  }
  B[j] = -999;
}
```

4. Write the control signals when the following instructions are executed in the single-cycle implementation of Y86.

```
0x2000: rmmov %rax, 0x100(%rcx)
        add   %rcx, %rdx
        push  %rdx
```

`rmmov %rax, 0x100(%rcx)`

Control Signal	Value	Control Signal	Value
needRegids	1	*needValC*	1
regA	10	*regB*	10
regE	00	*regM*	00
aluA	10	*aluB*	1
aluF	00	*setCond*	0
memRead	0	*memWrite*	1
memAddr	0	*memData*	0
newPC	00		

`add %rcx, %rdx`

Control Signal	Value	Control Signal	Value
needRegids	1	*needValC*	0
regA	10	*regB*	10
regE	10	*regM*	00
aluA	11	*aluB*	1
aluF	00	*setCond*	1
memRead	X	*memWrite*	0
memAddr	X	*memData*	X
newPC	00		

`push %rdx`

Control Signal	Value	Control Signal	Value
needRegids	1	*needValC*	0
regA	10	*regB*	01
regE	01	*regM*	00
aluA	00	*aluB*	1
aluF	00	*setCond*	0
memRead	0	*memWrite*	1
memAddr	0	*memData*	0
newPC	00		

5. Suppose we want to add a new instruction `imrmov` that copies 6-byte value from a memory address specified by a constant to a register.

   For example, `imrmov 0x8004, %rax` copies the 8-byte value at address `0x8004` to register `%rax`.

   (a) Design the format of this instruction type.

   `imrmov D, rA` can be represented in machine code using a format similar to `irmov` instruction, i.e. `|icode|ifun|rA|F|D|` Here, we use `rA` instead of `rB` because of the link from the data memory to the register file.

   (b) Design the computation steps for this instruction type.

Fetch	$icode, ifun$	$icode{:}ifun \leftarrow M_1[PC]$
	rA, rB	$rA{:}rB \leftarrow M_1[PC+1]$
	$valC$	$valC \leftarrow M_8[PC+2]$
	$valP$	$valP \leftarrow PC + 10$
Decode	$valA, srcA$	
	$valB, srcB$	
Execute	$valE$	$valE \leftarrow 0 + valC$
	$cond$	
Memory	$valM$	$valM \leftarrow M_8[valE]$
Write back	$dstE$	
	$dstM$	$R[rA] \leftarrow valM$
PC update	PC	$PC \leftarrow valP$

   (c) Design the control signals for this instruction for the single-cycle implementation.

Control Signal	Value	Control Signal	Value
$needRegids$	`1`	$needValC$	`1`
$regA$	`00`	$regB$	`00`
$regE$	`00`	$regM$	`10`
$aluA$	`10`	$aluB$	`0`
$aluF$	`00`	$setCond$	`0`
$memRead$	`1`	$memWrite$	`0`
$memAddr$	`0`	$memData$	`X`
$newPC$	`00`		

6. Write the truth table for the `memData` control signal.

$instruction$	$icode$	$memData$
`hlt`	`0000`	`X`
`nop`	`0001`	`X`
`rrmov`	`0010`	`X`
`irmov`	`0011`	`X`
`rmmov`	`0100`	`0`
`mrmov`	`0101`	`X`
`OP`	`0110`	`X`
`jXX`	`0111`	`X`
`call`	`1000`	`1`
`ret`	`1001`	`X`
`push`	`1010`	`0`
`pop`	`1011`	`X`
	`1100`	`X`
	`1101`	`X`
	`1110`	`X`
	`1111`	`X`