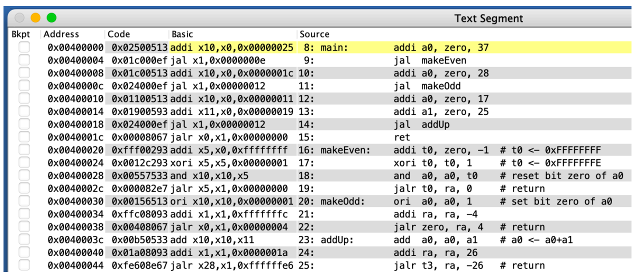
Question 5 (30 points):



Bkpt	Address	Code	Basic	Source		
	0x00400000	0x02500513	addi x10,x0,0x00000025	8: main:	addi a0, zero, 37	
	0x00400004	0x01c000ef	jal x1,0x0000000e	9:	jal makeEven	
	0x00400008	0x01c00513	addi x10,x0,0x0000001c	10:	addi a0, zero, 28	
	0x0040000c	0x024000ef	jal x1,0x00000012	11:	jal makeOdd	
	0x00400010	0x01100513	addi x10,x0,0x00000011	12:	addi a0, zero, 17	
	0x00400014	0x01900593	addi x11,x0,0x00000019	13:	addi a1, zero, 25	
	0x00400018	0x024000ef	jal x1,0x00000012	14:	jal addUp	
	0x0040001c	0x00008067	jalr x0,x1,0x00000000	15:	ret	
	0x00400020	0xfff00293	addi x5,x0,0xffffffff	16: makeEven:	addi t0, zero, -1	# t0 <- 0xFFFFFFFF
	0x00400024	0x0012c293	xori x5,x5,0x00000001	17:	xori t0, t0, 1	# t0 <- 0xFFFFFFFE
	0x00400028	0x00557533	and x10,x10,x5	18:	and a0, a0, t0	# reset bit zero of a0
	0x0040002c	0x000082e7	jalr x5,x1,0x00000000	19:	jalr t0, ra, 0	# return
	0x00400030	0x00156513	ori x10,x10,0x00000001	20: makeOdd:	ori a0, a0, 1	# set bit zero of a0
	0x00400034	0xffc08093	addi x1,x1,0xfffffffc	21:	addi ra, ra, -4	
	0x00400038	0x00408067	jalr x0,x1,0x00000004	22:	jalr zero, ra, 4	# return
	0x0040003c	0x00b50533	add x10,x10,x11	23: addUp:	add a0, a0, a1	# a0 <- a0+a1
	0x00400040	0x01a08093	addi x1,x1,0x0000001a	24:	addi ra, ra, 26	
	0x00400044	0xfe608e67	jalr x28,x1,0xffffffe6	25:	jalr t3, ra, -26	# return

(a) RARS Screenshot

Address	Binary Code	callVec	returnVec
0x1000103C	0xFFFFFFFF		
0x1000103C	0x01a08093	0	1
0x10001038	0x01a08093	0	0
0x1000103C	0x00b50533	0	0
0x10001038	0x00408067	0	1
0x10001034	0xffc08093	0	0
0x10001030	0x00156513	0	0
0x1000102C	0x000082e7	0	1
0x10001028	0x00557533	0	0
0x10001024	0x0012c293	0	0
0x10001020	0xfff00293	0	0
0x1000101C	0x00008067	0	1
0x10001018	0x024000ef	1	0
0x10001014	0x01900593	0	0
0x10001010	0x01100513	0	0
0x1000100C	0x024000ef	1	0
0x10001008	0x01c00513	0	0
0x10001004	0x01c000ef	1	0
0x10001000	0x02500513	0	0

(b) Binary Code stored in memory, vectors

Figure 1: RARS screenshot of the `RAS-Example.s` program. Unusual forms of return instructions are used to illustrate that any `jalr` instruction with source register `ra` is regarded as a return instruction. (a) In RARS the lower memory addresses are at the top. (b) In the memory representation our convention is that the lower addresses are at the bottom of the page.

The binary codes of the instructions of a RISC-V program can be stored in an array in memory. For example, Figure 1(a) has a RARS screenshot of a sample program. Assume that the binary representation of this program is stored in memory starting at the address `0x10001000` as shown in Figure 1(b). Figure 1(b) also shows two binary vectors `callVec` and `returnVec` that mark which instructions are a function call and which instructions are return statements. Binary vectors like these are useful when building a simulator. Later passes through the code can simply inspect the values in these binary vectors to decide if an action corresponding to a call or a return statement should be taken.

Write RISC-V assembly code for the function `RAS`. It receives as an argument the address of the first instruction of a RISC-V program. This program has been assembled and starting at that address in memory you find the binary code for the instructions. The end of the program is signalled by the sentinel word `0xFFFFFFFF`. Your `RAS` function does the following:

- sets to 1 the bits in the `callVec` that correspond to a function call instruction
- sets to 1 the bits in the `returnVec` that correspond to a return statement
- returns the number of function calls and the number of return statements found in the program.

Assume that all the bits in the `callVec` and `returnVec` binary vectors are 0 prior to the calling of the function `RAS`. The bit vectors `callVec` and `returnVec` are long enough to contain one bit for each instruction in the program.

The existing bit vector library makes available a `setBit` function with the following specification:

- Arguments:
  - `a0`: address of a bit vector
  - `a1`: an index into the bit vector specifying a bit
- Effect:
  - the bit specified is set

`RAS` must call the function `CallReturn` to determine if a given instruction is either a function call or a return statement. It must call the function `SetBit` (not shown) to set a bit in the bit vector.

- Arguments:
  - `a0`: address of first instruction in the program
  - `a1`: address of bit vector `callVec`
  - `a2`: address of bit vector `returnVec`
- Return Value:
  - `a0`: number of function call instructions found
  - `a1`: number of return statements found

```
RAS:
             addi sp, sp -32
             sw   s0, 0(sp)
             sw   s1, 4(sp)
             sw   s2, 8(sp)
             sw   s3, 12(sp)
             sw   s4, 16(sp)
             sw   s5, 20(sp)
             sw   s6, 24(sp)
             sw   ra, 28(sp)
             mv   s0, zero         # numCalls <- 0
             mv   s1, zero         # numReturns <- 0
             mv   s3, zero         # InstrCounter <-- 0
             mv   s2, a0           # InstrPointer <- address of first instruction
             mv   s5, a1           # callVec
             mv   s6, a2           # returnVec
             addi s4, zero, -1     # s4 <- Sentinel value
NextInstr:   lw   t0, 0(s2)        # t0 <- currentInstruction
             beq  t0, s4, ProgEnd  # if found sentinel
             mv   a0, s2
             jal  ra, CallReturn
             addi s2, s2, 4        # InstrPointer++
             addi s3, s3, 1        # InstrCounter++
             beqz a0, NextInstr
             addi a1, s3, -1       # a1 <- CurrentInstruction position
             beq  a0, s4, IsReturn # if CallReturn set a0 == -1
             mv   a0, s5           # a0 <- callVec
             jal  setBit
             addi s0, s0, 1        # numCalls++
             j    NextInstr
IsReturn:    mv   a0, s6           # a0 <- returnVec
             jal  setBit
             addi s1, s1, 1        # numReturns++
             j    NextInstr
ProgEnd:     mv   a0, s0           # a0 <- numCalls
             mv   a1, s1           # a1 <- numReturns
             lw   s0, 0(sp)
             lw   s1, 4(sp)
             lw   s2, 8(sp)
             lw   s3, 12(sp)
             lw   s4, 16(sp)
             lw   s5, 20(sp)
             lw   s6, 24(sp)
             lw   ra, 28(sp)
             addi sp, sp, 32
             jalr x0, ra, 0
```

Figure 2: A solution for `RAS`.