Group - 8
Suhas Jain (19CS30048)
Monal Prasad (19CS30030)

BUBBLE SORT

To test the correct working of the ISA of our KGP-RISC processor, we wrote the code to sort 10 integers using bubble sort. All the necessary files and simulation results have been attached in the folder as well as attached and explained below.

Bubble sort code:

```
main:
   xor $20, $20         # $20 = 0 (base address)
   xor $21, $21         # $21 = 0 (n)
   addi $21, 10         # $21 = n = 10
   xor $8, $8           # $8 = 0 (outer loop variable (i))
   xor $9, $9           # $9 = 0 (inner loop variable (j))

fori:
   xor $10, $10         # label 20
   add $10, $8          # $10 = i
   comp $11, $21        # $11 = -n
   add $10, $11         # $10 = i - n
   addi $10, 1          # $10 = i - (n - 1) = i - n + 1
   bz $10, exitfori     # if i = n-1, end outer loop
   xor $9, $9           # j = 0

forj:
   xor $11, $11         # $11 = 0
   add $11, $9          # $11 = j
   add $11, $10         # $11 = j + i - n + 1
   bz $11, exitforj     # if j = n - i - 1, end inner loop

   xor $12, $12         # $12 = 0
   add $12, $9          # $12 = j
   shll $12, 2          # $12 = 4 * j
   add $12, $20         # $12 =  arr + 4 * j
   lw $13, 0($12)       # $13 = arr[j]

   xor $4, $4           # $4 = 0
   add $4, $12          # $4 = arr + 4 * j
   addi $12, 4          # $4 = arr + 4 * j + 4
   lw $14, 0($12)       # $14 = arr[j + 1]
```

```
   xor $5, $5          # $5 = 0
   add $5, $12         # $5 = arr + 4 * j
   comp $15, $14       # $5 = -arr[j+1]
   add $13, $15        # $13 = arr[j] - arr[j + 1]
   bltz $13, incj      # don't swap if arr[j] > arr[j + 1]
   bz $13, incj        # don't swap if arr[j] = arr[j + 1]
   bl swap             # swap otherwise

incj:
   addi $9, 1          # j = j + 1
   b forj              # continue inner loop

swap:
   lw $18, 0($4)       # $18 = arr[j]
   lw $19, 0($5)       # $19 = arr[j + 1]
   sw $18, 0($5)       # arr[j+1] = $18 (save in swpped location)
   sw $19, 0($4)       # arr[j] = $19 (save in swpped location)
   br $31

exitforj:
   addi $8, 1          # i = i + 1
   b fori              # continue inner loop

exitfori:
   xor $16, $16        # $16 = 0
   addi $16, 1         # $16 = 1 (flag indication completion of execution)
```

COE file with binary instructions (to be loaded in instruction memory):

```
memory_initialization_radix=2;
memory_initialization_vector=
00000010100101000000000000000011
00000010101101010000000000000011
00000110101000000000000000001010
00000001000010000000000000000011
00000001001010010000000000000011
00000001010010100000000000000011
00000001010010000000000000000000
00000001011101010000000000000001
00000001010010110000000000000000
00000101010000000000000000000001
00011101010000000000000000011110
00000001001010010000000000000011
00000001011010110000000000000011
00000001011010010000000000000000
00000001011010100000000000000000
00011101011000000000000000010111
```

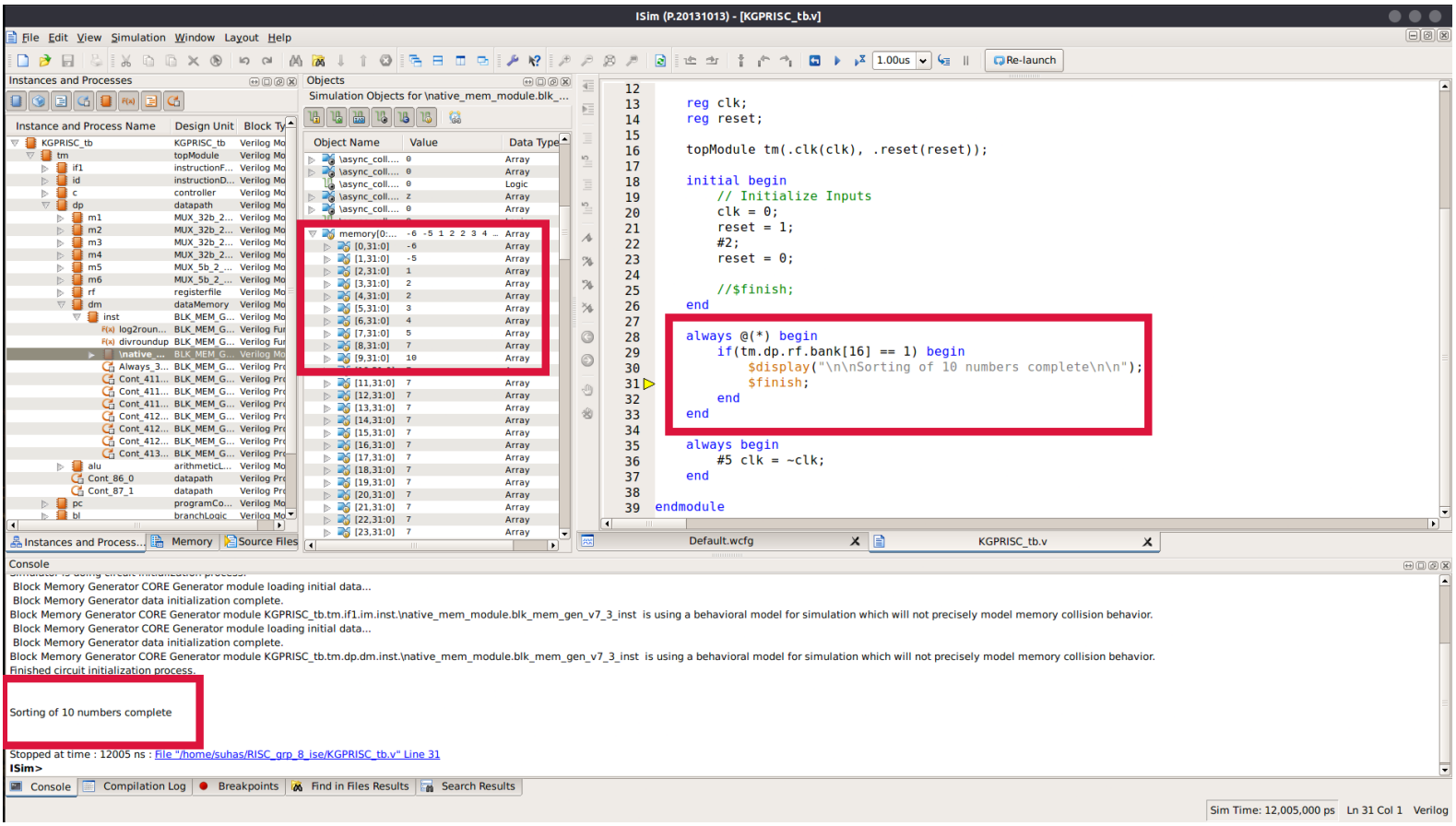
```
00000001100011000000000000000011
00000001100010010000000000000000
00000001100000000001000000000100
00000001100101000000000000000000
00001101100011010000000000000000
00000000100001000000000000000011
00000000100011000000000000000000
00000101100000000000000000000100
00001101100011100000000000000000
00000000101001010000000000000011
00000000101011000000000000000000
00000001111011100000000000000001
00000001101011110000000000000000
00011001101000000000000000000010
00011101101000000000000000000001
00101000000000000000000000000010
00000101001000000000000000000001
00100100000111111111111111101010
00001100100100100000000000000000
00001100101100110000000000000000
00010000101100100000000000000000
00010000100100110000000000000000
00010111111000000000000000000000
00000101000000000000000000000001
00100100000111111111111111011100
00000010000100000000000000000011
00000110000000000000000000000001
```

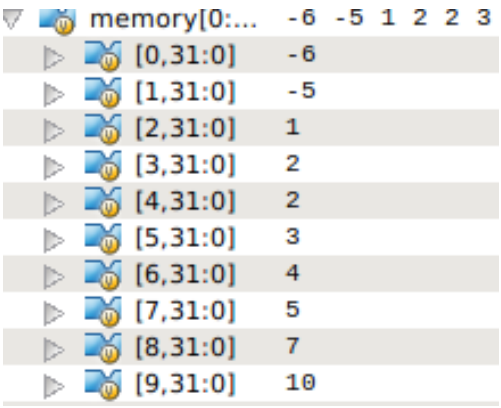
COE file with 10 integers to be sorted (to be loaded in data memory):

```
memory_initialization_radix=10;
memory_initialization_vector=
10
5
-6
3
2
-5
4
1
2
7
```

Simulation Results:

Simulation results of the bubble sort algorithm have been attached below. It can be noticed that the 10 integer values which included both positive and negative numbers which were loaded in the memory in the unsorted order have now been sorted and appear on index 0-9 in memory in the correct order. Also we print a message to detect the completion of execution ($16 register used as flag), that also works perfectly as expected.





ITERATIVE FIBONACCI CALCULATION

To test the correct working of the ISA of our KGP-RISC processor the second test includes calculation of the nth fibonacci number using iteration. All the necessary files and simulation results have been attached in the folder as well as attached and explained below. In this example we have computed the 6th fibonacci number.

Iterative Fibonacci Code:

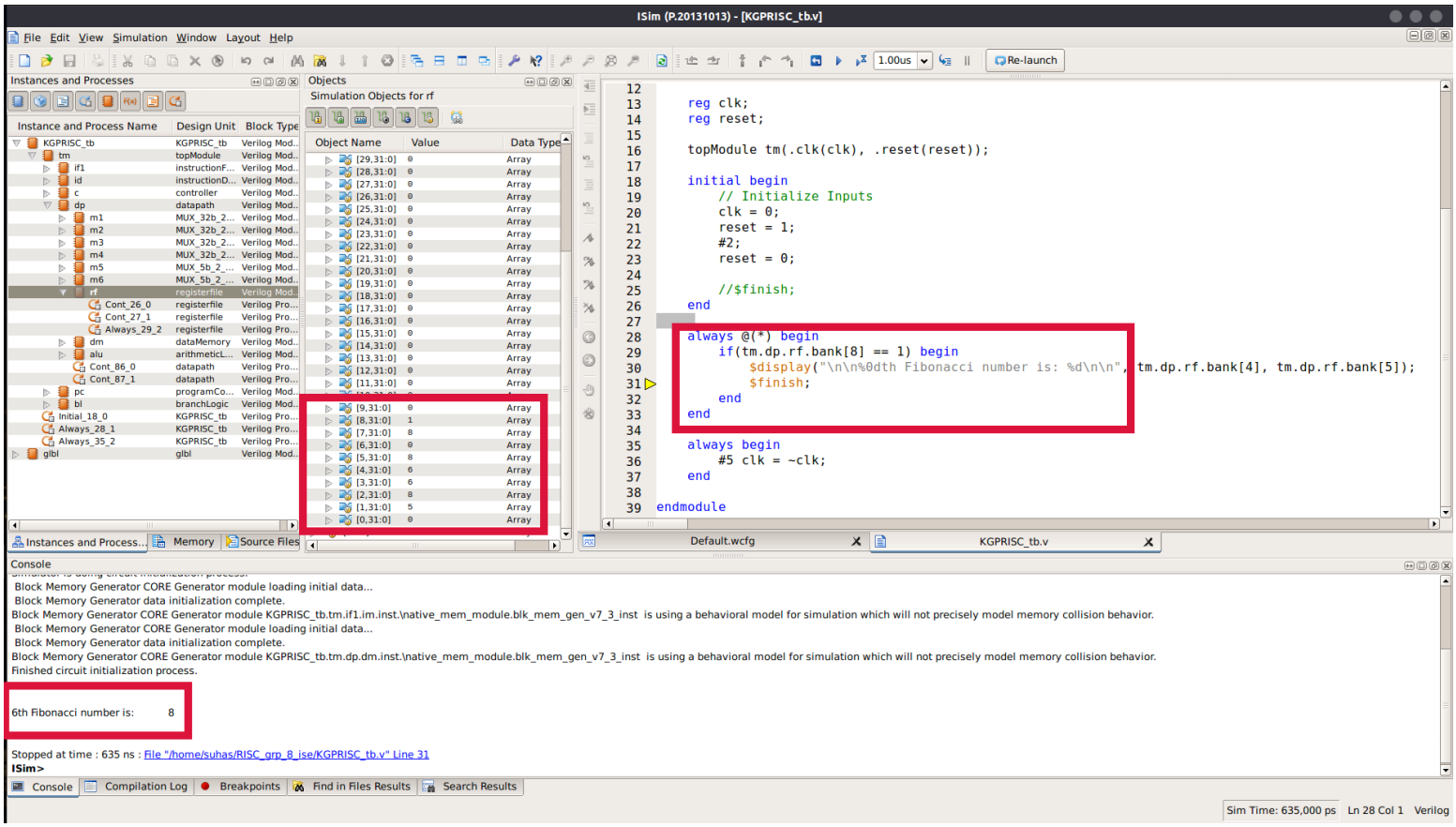
```
main:
   xor $1, $1       # $1 = 0 (previous element)
   xor $2, $2       # $2 = 0 (current element)
   xor $3, $3       # $3 = 0 (loop variable (i) )
   xor $4, $4       # $4 = 0 (n)
   xor $5, $5       # $5 = 0 (final answer)
   addi $1, 1       # $1 = 1 (first element)
   addi $2, 1       # $2 = 1 (second element)
   addi $3, 2       # i = 2 (as we already have 2 elements)
   addi $4, 6       # n = 6

fib:
   comp $6, $4      # $6 = - n
   add $6, $3       # $6 = i - n
   bz $6, endfib    # $6 = 0  ==> i = n, end loop

   add $1, $2       # $1 = $1 + $2
   xor $7, $7       # $7 = 0
   add $7, $1       # $7 = $1
   xor $1, $1       # $1 = 0
   add $1, $2       # $1 = $2
   xor $2, $2       # $2 = 0
   add $2, $7       # $2 = $7

   addi $3, 1       # i = i + 1
   b fib            # continue loop

endfib:
   add $5, $2       # Final answer = $2
   xor $8, $8       # $8 = 0 (completion flag)
   addi $8, 1       # $8 = 1
```

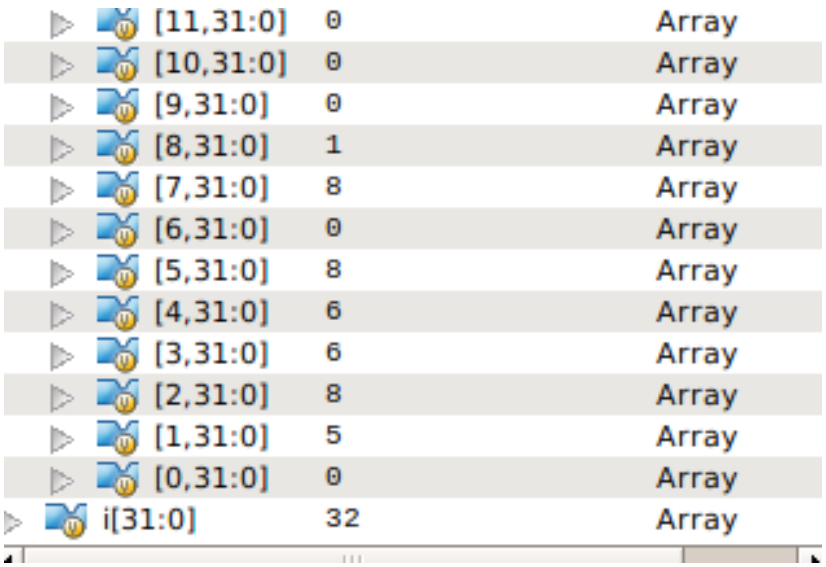
COE file with binary instructions (to be loaded in instruction memory):

```
memory_initialization_radix=2;
memory_initialization_vector=
00000000001000010000000000000011
00000000010000100000000000000011
00000000011000110000000000000011
00000000100001000000000000000011
00000000101001010000000000000011
00000100001000000000000000000001
00000100010000000000000000000001
00000100011000000000000000000010
00000100100000000000000000000110
00000000110001000000000000000001
00000000110000110000000000000000
00011100110000000000000000001110
00000100110000001111111111111111
00011100110000000000000000001001
00000000001000100000000000000000
00000000111001110000000000000011
00000000111000010000000000000000
00000000001000010000000000000011
00000000001000100000000000000000
00000000010000100000000000000011
00000000010001110000000000000000
00000100011000000000000000000001
00100100000111111111111111110010
00000000101000010000000000000000
00000001000010000000000000000011
00000101000000000000000000000001
00000000101000100000000000000000
00000001000010000000000000000011
00000101000000000000000000000001
```

Simulation Results:

Simulation results of the iterative fibonacci algorithm have been attached below. It can be noticed that the values in the register $5 stores the final values (which is 8) and register $4 stores the value of n (which is 6). Also we print a message to detect the completion of execution ($8 register used as flag), that also works perfectly as expected.





RECURSIVE FIBONACCI CALCULATION

To test the correct working of the ISA of our KGP-RISC processor the third and last test includes calculation of the nth fibonacci number using recursion. All the necessary files and simulation results have been attached in the folder as well as attached and explained below. In this example we have computed the 7th fibonacci number.

Recursive Fibonacci Code:

```
main:
   xor $29, $29         # $29 = sp = 0 (stack pointer)
   xor $1, $1           # finish flag set to 0
   xor $4, $4           # $4 = n = 0
   xor $3, $3           # $3 = n = 0 (won't change)
   addi $4, 6           # $4 = n = 6
   add $3, $4           # $3 = n = 6
   bl fib               # branch and link to fib
   addi $1, 1           # flag set to 1 on completion

fib:
   addi $29, 12         # sp = sp + 12 (make space in stack)
   sw $31, -12($29)     # store return address on stack
   sw $4, -8($29)       # store n onto stack
   compi $10, 2         # $10 = -2
   add $10, $4          # $10 = n-2
   bltz $10, endfib     # if n-2 < 0, jump to endfib
   addi $4, -1          # $4 = n-1
   bl fib               # branch and link to fib
   sw $2, -4($29)       # store fib(n-1) in memory

   lw $4, -8($29)       # load n from memory into $4
   addi $4, -2          # $4 = n - 2
   bl fib               # branch and link to fib
   lw $8, -4($29)       # load fib(n-1) into $t0
   add $2, $8           # $2  = fib(n-1) + fib(n-2) (return value)
   b end

endfib:
   xor $2, $2           # v0 = 0
   add $2, $4           # v0 = n (if n == 0 || n == 1)
end:
   lw $31 -12($29)      # $31 = return address
   addi $29, -12        # restore stack pointer
   br $31
```

COE file with binary instructions (to be loaded in instruction memory):

```
memory_initialization_radix=2;
memory_initialization_vector=
00000011101111010000000000000011
00000000001000010000000000000011
00000000100001000000000000000011
00000000011000110000000000000011
00000100100000000000000000000111
00000000011001000000000000000000
00101000000000000000000000000001
00000100001000000000000000000001
00000111101000000000000000001100
00010011101111111111111111110100
00010011101001001111111111111000
00001001010000000000000000000010
00000001010001000000000000000000
00011001010000000000000000001001
00000100100000001111111111111111
00101000000111111111111111111000
00010011101000101111111111111100
00001111101001001111111111111000
00000100100000001111111111111110
00101000000111111111111111110100
00001111101010001111111111111100
00000000010010000000000000000000
00100100000000000000000000000010
00000000010000100000000000000011
00000000010001000000000000000000
00001111101111111111111111110100
00000111101000001111111111110100
00010111111000000000000000000000
```

Simulation Results:

Simulation results of the recursive fibonacci algorithm have been attached below. It can be noticed that the values in the register $2 stores the final values (which is 13) and register $3 stores the value of n (which is 7). Also we print a message to detect the completion of execution ($1 register used as flag), that also works perfectly as expected.