

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
00011101010000000000000000001110
00000001001010010000000000000011
00000001011010110000000000000011
00000001011010010000000000000000
00000001011010100000000000000000
00011101011000000000000000010111

```

```
0000000110001100000000000000011
0000000110001001000000000000000
00000001100000000001000000000100
0000000110010100000000000000000
0000110110001101000000000000000
0000000010000100000000000000011
0000000010001100000000000000000
0000010110000000000000000000100
0000110110001110000000000000000
0000000010100101000000000000011
0000000010101100000000000000000
0000000111101110000000000000001
0000000110101111000000000000000
000110011010000000000000000010
000111011010000000000000000001
001010000000000000000000000010
000001010010000000000000000001
0010010000011111111111111101010
0000110010010010000000000000000
0000110010110011000000000000000
0001000010110010000000000000000
0001000010010011000000000000000
0001011111100000000000000000000
000001010000000000000000000001
0010010000011111111111111101100
000000100001000000000000000011
000001100000000000000000000001
```

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.

The screenshot displays the ISim (P.20131013) - [KGPRISC_tb.v] window. The left pane shows the 'Instances and Processes' tree, with the 'memory' block highlighted. The middle pane shows the 'Simulation Objects for \'native_mem_module.blk_...\'' table, which lists memory locations and their values. The right pane shows the Verilog code, with the completion message highlighted. The bottom pane shows the console output, which includes the completion message and the final sorted array.

Object Name	Value	Data Type
memory[0:...	-6 -5 1 2 2 3	Array
[0,31:0]	-6	Array
[1,31:0]	-5	Array
[2,31:0]	1	Array
[3,31:0]	2	Array
[4,31:0]	2	Array
[5,31:0]	3	Array
[6,31:0]	4	Array
[7,31:0]	5	Array
[8,31:0]	7	Array
[9,31:0]	10	Array

```
12
13 reg clk;
14 reg reset;
15
16 topModule tm(.clk(clk), .reset(reset));
17
18 initial begin
19     // Initialize Inputs
20     clk = 0;
21     reset = 1;
22     #2;
23     reset = 0;
24
25     // $finish;
26 end
27
28 always @(*) begin
29     if(tm.dp.rf.bank[16] == 1) begin
30         $display("\n\nSorting of 10 numbers complete\n\n");
31         $finish;
32     end
33 end
34
35 always begin
36     #5 clk = ~clk;
37 end
38
39 endmodule
```

Sorting of 10 numbers complete

Stopped at time : 12005 ns : file "/home/suhas/RISC_grp_8_ise/KGPRISC_tb.v" line 31

ISim>

Console | Compilation Log | Breakpoints | Find in Files Results | Search Results

Sim Time: 12,005,000 ps Ln 31 Col 1 Verilog

memory[0:...	-6 -5 1 2 2 3
[0,31:0]	-6
[1,31:0]	-5
[2,31:0]	1
[3,31:0]	2
[4,31:0]	2
[5,31:0]	3
[6,31:0]	4
[7,31:0]	5
[8,31:0]	7
[9,31:0]	10

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=
000000000100010000000000000011
000000000100010000000000000011
000000000110011000000000000011
000000001000100000000000000011
000000001010010100000000000011
00001000100000000000000000001
00001000100000000000000000001
00001000110000000000000000010
00001001000000000000000000110
00000000110010000000000000001
00000000110001100000000000000
00011100110000000000000001110
00001001100000111111111111111
000111001100000000000000001001
00000000010010000000000000000
000000001110011100000000000011
00000000111000100000000000000
0000000001000100000000000011
00000000010010000000000000000
00000000010011100000000000000
00001000110000000000000000001
001001000011111111111111110010
00000000101000100000000000000
0000000100010000000000000011
00001010000000000000000000001
00000000101001000000000000000
0000000100010000000000000011
00001010000000000000000000001
```

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.

The screenshot displays the ISim (P.20131013) - [KGPRISC_tb.v] simulation environment. The Verilog code on the right includes a module with registers for clock, reset, and a Fibonacci counter. The Object browser in the center shows the 'rf' register file, with registers \$0 through \$31. The Console on the left shows the output of the simulation, including the message '6th Fibonacci number is: 8'.

```
12 reg clk;
13 reg reset;
14
15 topModule tm(.clk(clk), .reset(reset));
16
17 initial begin
18     // Initialize Inputs
19     clk = 0;
20     reset = 1;
21     #2;
22     reset = 0;
23
24     //$finish;
25 end
26
27 always @(*) begin
28     if(tm.dp.rf.bank[8] == 1) begin
29         $display("\n\n%0dth Fibonacci number is: %d\n\n", tm.dp.rf.bank[4], tm.dp.rf.bank[5]);
30         $finish;
31     end
32 end
33
34 always begin
35     #5 clk = ~clk;
36 end
37
38 endmodule
39
```

Simulation Objects for rf

Object Name	Value	Data Type
[29,31:0]	0	Array
[28,31:0]	0	Array
[27,31:0]	0	Array
[26,31:0]	0	Array
[25,31:0]	0	Array
[24,31:0]	0	Array
[23,31:0]	0	Array
[22,31:0]	0	Array
[21,31:0]	0	Array
[20,31:0]	0	Array
[19,31:0]	0	Array
[18,31:0]	0	Array
[17,31:0]	0	Array
[16,31:0]	0	Array
[15,31:0]	0	Array
[14,31:0]	0	Array
[13,31:0]	0	Array
[12,31:0]	0	Array
[11,31:0]	0	Array
[10,31:0]	0	Array
[9,31:0]	0	Array
[8,31:0]	1	Array
[7,31:0]	8	Array
[6,31:0]	0	Array
[5,31:0]	8	Array
[4,31:0]	6	Array
[3,31:0]	6	Array
[2,31:0]	8	Array
[1,31:0]	5	Array
[0,31:0]	0	Array

Console

```
Block Memory Generator CORE Generator module loading initial data...
Block Memory Generator data initialization complete.
Block Memory Generator CORE Generator module KGPRISC_tb.tm.if1.inst.native_mem_module.blk_mem_gen_v7_3_inst is using a behavioral model for simulation which will not precisely model memory collision behavior.
Block Memory Generator CORE Generator module loading initial data...
Block Memory Generator data initialization complete.
Block Memory Generator CORE Generator module KGPRISC_tb.tm.dp.dm.inst.native_mem_module.blk_mem_gen_v7_3_inst is using a behavioral model for simulation which will not precisely model memory collision behavior.
Finished circuit initialization process.

6th Fibonacci number is: 8

Stopped at time : 635 ns : File "/home/suhas/RISC_gpr_8_ise/KGPRISC_tb.v" Line 31
ISim>
```

Sim Time: 635,000 ps Ln 28 Col 1 Verilog

[11,31:0]	0	Array
[10,31:0]	0	Array
[9,31:0]	0	Array
[8,31:0]	1	Array
[7,31:0]	8	Array
[6,31:0]	0	Array
[5,31:0]	8	Array
[4,31:0]	6	Array
[3,31:0]	6	Array
[2,31:0]	8	Array
[1,31:0]	5	Array
[0,31:0]	0	Array
i[31:0]	32	Array

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)

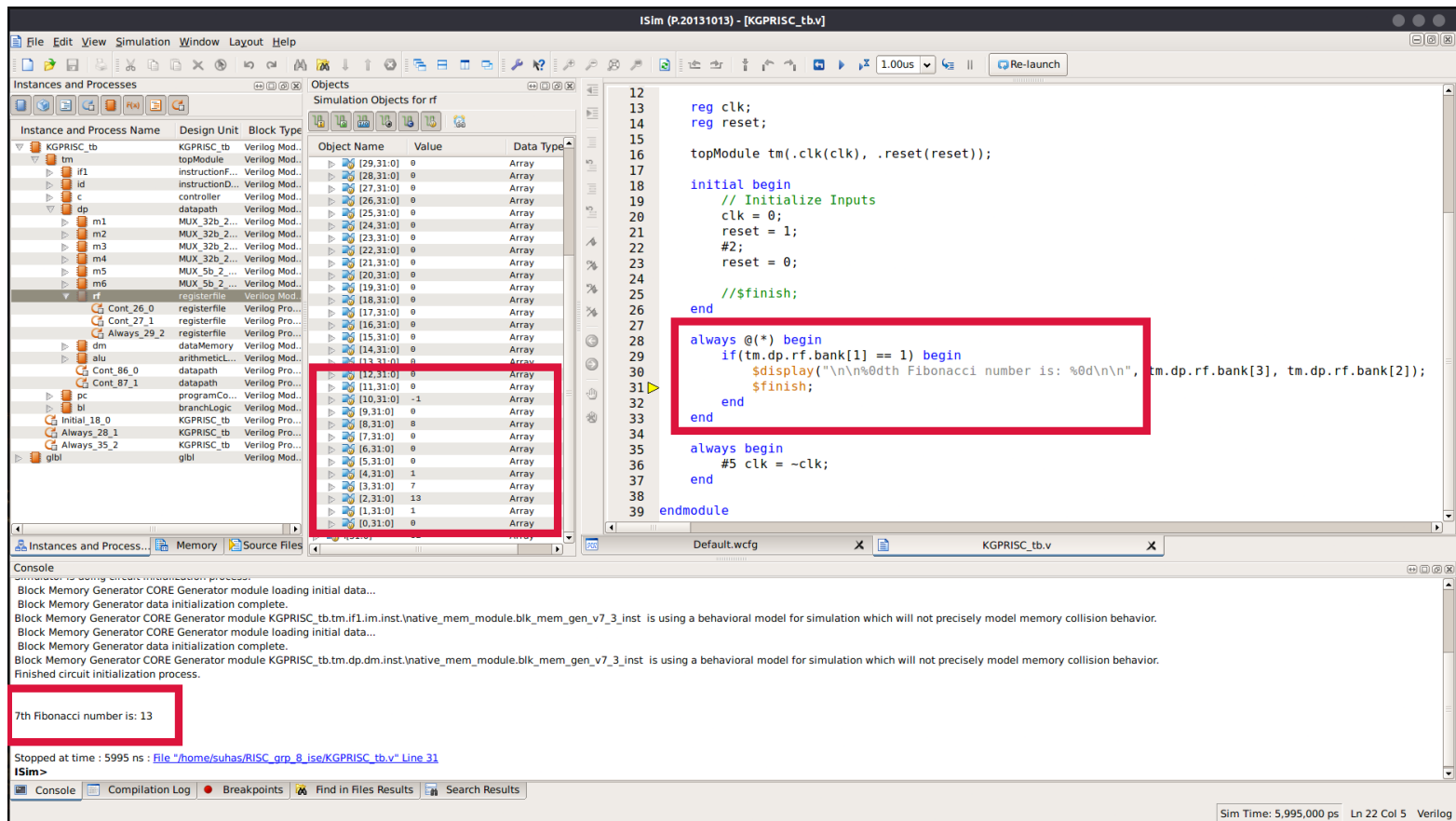
end:
    lw $31, -12($29)  # $31 = return address
    addi $29, -12     # restore stack pointer
    br $31
```


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

```
memory_initialization_radix=2;
memory_initialization_vector=
0000001110111101000000000000011
0000000000100001000000000000011
0000000010000100000000000000011
0000000001100011000000000000011
0000010010000000000000000000111
0000000001100100000000000000000
0010100000000000000000000000001
0000010000100000000000000000001
0000011110100000000000000001100
0001001110111111111111111110100
0001001110100100111111111111000
000010010100000000000000000010
0000000101000100000000000000000
0001100101000000000000000001001
0000010010000000111111111111111
001010000001111111111111111000
000100111010001011111111111100
000011111010010011111111111000
000001001000000011111111111110
001010000001111111111111110100
000011111010100011111111111100
000000000100100000000000000000
001001000000000000000000000010
000000000100001000000000000011
000000000100010000000000000000
000011111011111111111111110100
000001111010000011111111110100
000101111110000000000000000000
```

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.



Simulation Objects for rf

Object Name	Value	Data Type
[29,31:0]	0	Array
[28,31:0]	0	Array
[27,31:0]	0	Array
[26,31:0]	0	Array
[25,31:0]	0	Array
[24,31:0]	0	Array
[23,31:0]	0	Array
[22,31:0]	0	Array
[21,31:0]	0	Array
[20,31:0]	0	Array
[19,31:0]	0	Array
[18,31:0]	0	Array
[17,31:0]	0	Array
[16,31:0]	0	Array
[15,31:0]	0	Array
[14,31:0]	0	Array
[13,31:0]	0	Array
[12,31:0]	0	Array
[11,31:0]	0	Array
[10,31:0]	-1	Array
[9,31:0]	0	Array
[8,31:0]	8	Array
[7,31:0]	0	Array
[6,31:0]	0	Array
[5,31:0]	0	Array
[4,31:0]	1	Array
[3,31:0]	7	Array
[2,31:0]	13	Array
[1,31:0]	1	Array
[0,31:0]	0	Array

Console

Block Memory Generator CORE Generator module loading initial data...

Block Memory Generator data initialization complete.

Block Memory Generator CORE Generator module KGPRISC_tb.tm.if1.im.inst.native_mem_module.blk_mem_gen_v7_3_inst is using a behavioral model for simulation which will not precisely model memory collision behavior.

Block Memory Generator CORE Generator module loading initial data...

Block Memory Generator data initialization complete.

Block Memory Generator CORE Generator module KGPRISC_tb.tm.dp.dm.inst.native_mem_module.blk_mem_gen_v7_3_inst is using a behavioral model for simulation which will not precisely model memory collision behavior.

Finished circuit initialization process.

7th Fibonacci number is: 13

Stopped at time : 5995 ns : File "/home/suhas/RISC_gpr_8_ise/KGPRISC_tb.v" Line 31

ISim>

Console | Compilation Log | Breakpoints | Find in Files Results | Search Results

Sim Time: 5,995,000 ps Ln 22 Col 5 Verilog

[12,31:0]	0	Array
[11,31:0]	0	Array
[10,31:0]	-1	Array
[9,31:0]	0	Array
[8,31:0]	8	Array
[7,31:0]	0	Array
[6,31:0]	0	Array
[5,31:0]	0	Array
[4,31:0]	1	Array
[3,31:0]	7	Array
[2,31:0]	13	Array
[1,31:0]	1	Array
[0,31:0]	0	Array
i[31:0]	32	Array