12/16/2022

Processor Architectures

Project: RISC-V Bubble sort

Jaani Söderström
2102304

Contents

1. Problem description 2

2. High level implementation 2

3. Low level implementation 3

4. Simulation results 5

First test: 5

Second test: 7

5. Inference 9

1. Problem description

The bubble sort algorithm sorts and array of elements by repeatedly iterating through the array and comparing adjacent elements. If the elements are in the wrong order, they are swapped. This process is repeated until the array is sorted.

Pseudocode of bubble sort:

```
bubble_sort(A)
repeat
  swapped = false
  for i = 1 to length(A) - 1
    if A[i] > A[i + 1]
      swap A[i] and A[i + 1]
      swapped = true
until not swapped
return A
```

2. High level implementation

The bubble sort algorithm in java:

```java
public static int[] bubble_sort(int[] A) {
    boolean swapped;
    do {
        swapped = false;
        // iterate through the array
        for (int i = 0; i < A.length - 1; i++) {
            // if the current element is greater than the next
element, swap them
            if (A[i] > A[i + 1]) {
                int temp = A[i];
                A[i] = A[i + 1];
                A[i + 1] = temp;
                swapped = true;
            }
        }
    } while (swapped);
    return A;
}
```

Java version of the algorithm uses temporary variable (int temp) to store the item because it is not possible to swap items in array in java, but the idea is still the same.

3. Low level implementation

RISC-V assembly code with comments:

```
#creating array and saving it to memory (you can change the values
manually)
addi x1,x0,5
sw x1,4(x0)
addi x1,x0,8
sw x1,8(x0)
addi x1,x0,3
sw x1,12(x0)
addi x1,x0,1
sw x1,16(x0)
addi x1,x0,4
sw x1,20(x0)

#save number corresponding to total number of instances to x4
addi x4, x0, 5
#save number to x6 which will be deleted from x4 to check if all cells are
sorted
addi x6, x0, 1

loop:
add x5, x0, x4
compare0:
#load [0] and [1] compare them
lw x1,4(x0)
lw x2,8(x0)
blt x1,x2,compare1
#if [0] > [1] swap places and remove x6 from x4 so check knows to repeat
the loop
sw x2,4(x0)
sw x1,8(x0)
sub x5, x4, x6

compare1:
#compare [1] and [2]
lw x1,8(x0)
lw x2,12(x0)
blt x1,x2,compare3
sw x2,8(x0)
sw x1,12(x0)
sub x5, x4, x6

compare3:
#compare [2] and [3]
lw x1,12(x0)
lw x2,16(x0)
blt x1,x2,compare4
sw x2,12(x0)
sw x1,16(x0)
sub x5, x4, x6

compare4:
#compare [1] and [2]
lw x1,16(x0)
lw x2,20(x0)
blt x1,x2,check
sw x2,16(x0)
sw x1,20(x0)
sub x5, x4, x6
#both operations go the check method but first one skips the swapping
#if those elements are already sorted

check:
#if x4 and x5 are equal jump to exit = array is sorted, else: repeat the
loop
beq x4,x5,exit
jal ra, loop

exit:
```

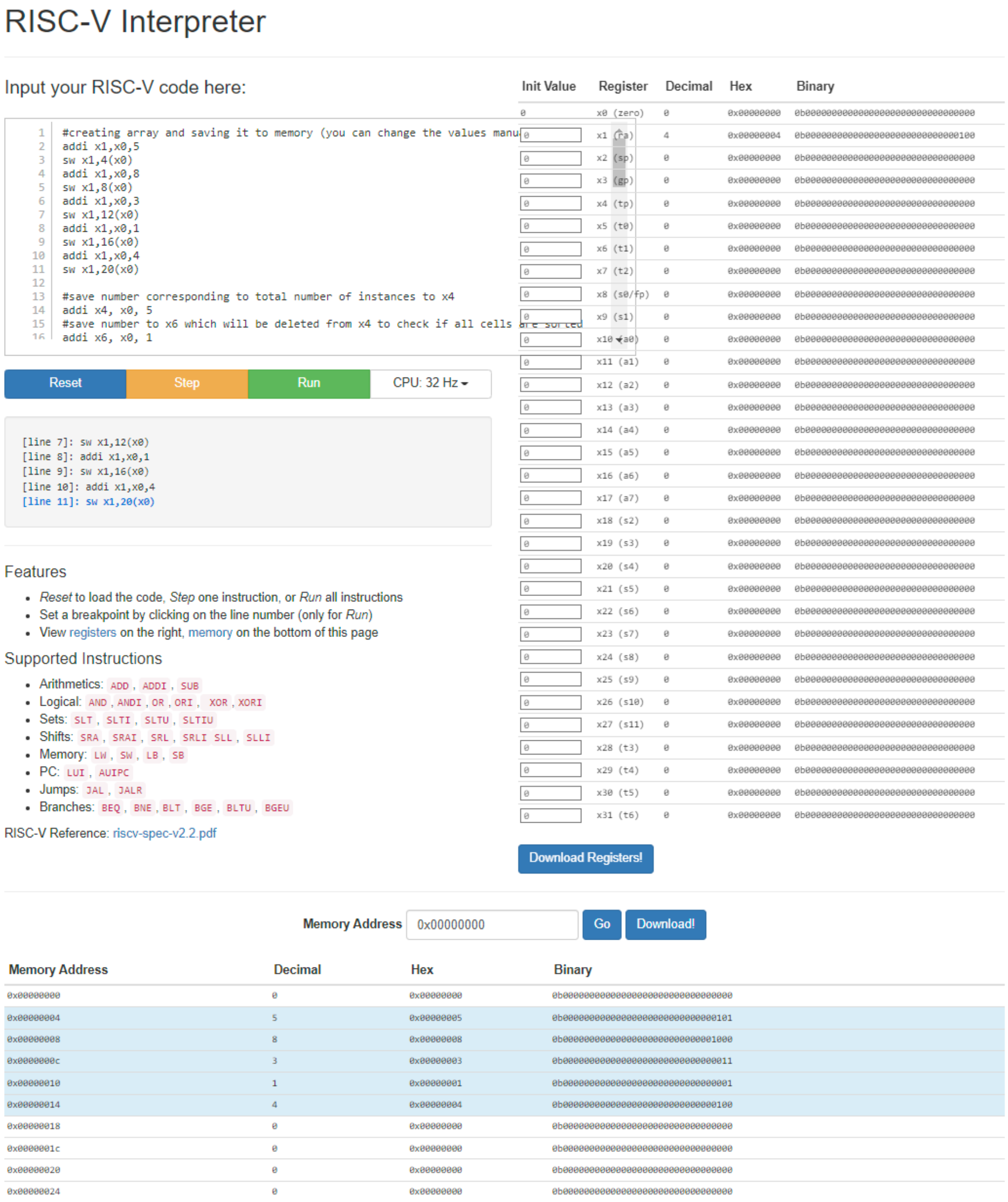
In the code I create array [5,8,3,1,4] and sort it with using:

https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/

Now algorithm can only sort arrays which have five items, but it can be edited in way to sort different number of items.

4. Simulation results

First test:



RISC-V Interpreter

Input your RISC-V code here:

```
#creating array and saving it to memory (you can change the values manu
addi x1,x0,5
sw x1,4(x0)
addi x1,x0,8
sw x1,8(x0)
addi x1,x0,3
sw x1,12(x0)
addi x1,x0,1
sw x1,16(x0)
addi x1,x0,4
sw x1,20(x0)

#save number corresponding to total number of instances to x4
addi x4, x0, 5
#save number to x6 which will be deleted from x4 to check if all cells are sorted
addi x6, x0, 1
```

```
[line 7]: sw x1,12(x0)
[line 8]: addi x1,x0,1
[line 9]: sw x1,16(x0)
[line 10]: addi x1,x0,4
[line 11]: sw x1,20(x0)
```

Features

- *Reset* to load the code, *Step* one instruction, or *Run* all instructions
- Set a breakpoint by clicking on the line number (only for *Run*)
- View registers on the right, memory on the bottom of this page

Supported Instructions

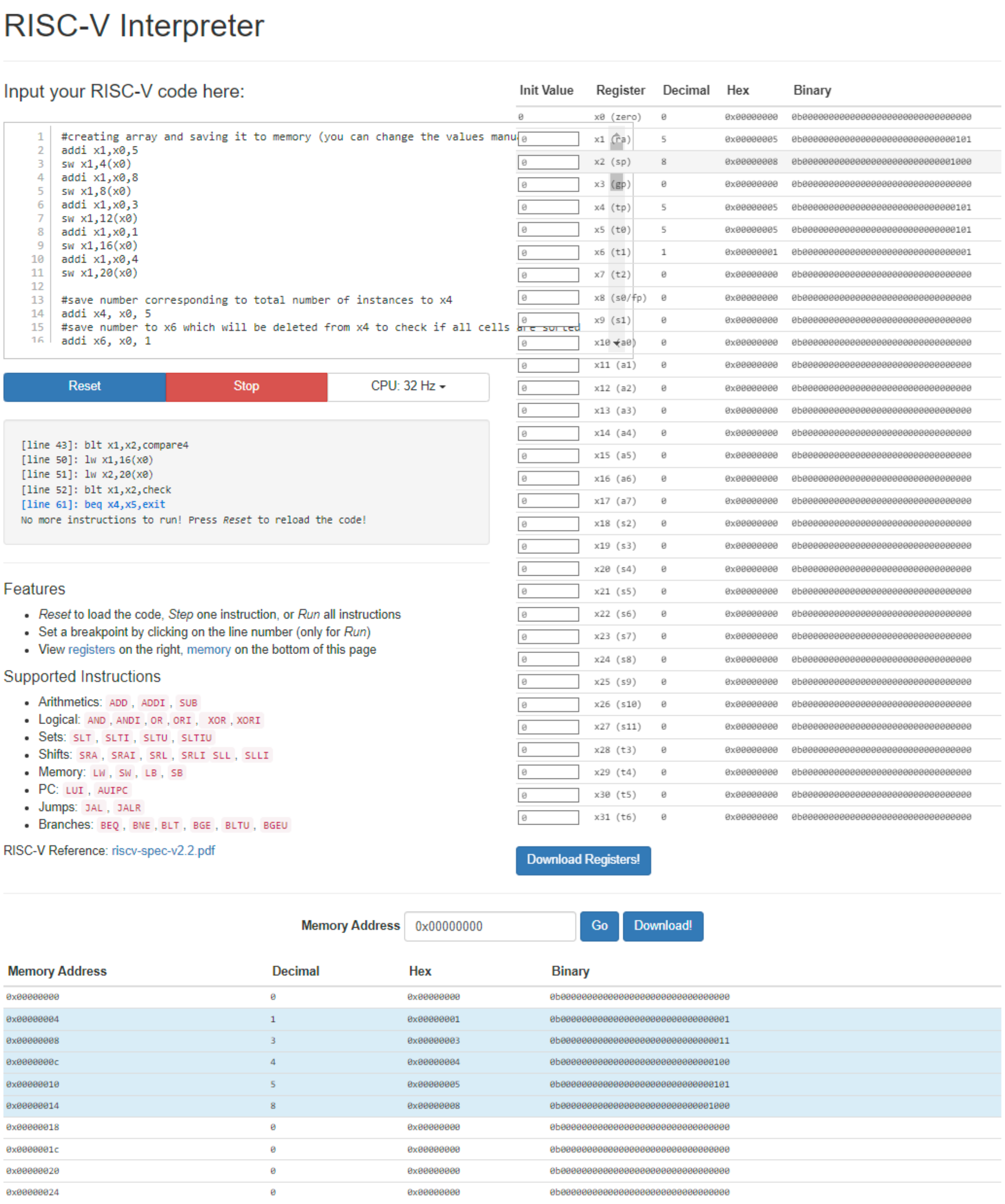
- Arithmetics: ADD , ADDI , SUB
- Logical: AND , ANDI , OR , ORI , XOR , XORI
- Sets: SLT , SLTI , SLTU , SLTIU
- Shifts: SRA , SRAI , SRL , SRLI SLL , SLLI
- Memory: LW , SW , LB , SB
- PC: LUI , AUIPC
- Jumps: JAL , JALR
- Branches: BEQ , BNE , BLT , BGE , BLTU , BGEU

RISC-V Reference: riscv-spec-v2.2.pdf

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
0	x1 (ra)	4	0x00000004	0b00000000000000000000000000000100
0	x2 (sp)	0	0x00000000	0b00000000000000000000000000000000
0	x3 (gp)	0	0x00000000	0b00000000000000000000000000000000
0	x4 (tp)	0	0x00000000	0b00000000000000000000000000000000
0	x5 (t0)	0	0x00000000	0b00000000000000000000000000000000
0	x6 (t1)	0	0x00000000	0b00000000000000000000000000000000
0	x7 (t2)	0	0x00000000	0b00000000000000000000000000000000
0	x8 (s0/fp)	0	0x00000000	0b00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0b00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	0b00000000000000000000000000000000
0	x12 (a2)	0	0x00000000	0b00000000000000000000000000000000
0	x13 (a3)	0	0x00000000	0b00000000000000000000000000000000
0	x14 (a4)	0	0x00000000	0b00000000000000000000000000000000
0	x15 (a5)	0	0x00000000	0b00000000000000000000000000000000
0	x16 (a6)	0	0x00000000	0b00000000000000000000000000000000
0	x17 (a7)	0	0x00000000	0b00000000000000000000000000000000
0	x18 (s2)	0	0x00000000	0b00000000000000000000000000000000
0	x19 (s3)	0	0x00000000	0b00000000000000000000000000000000
0	x20 (s4)	0	0x00000000	0b00000000000000000000000000000000
0	x21 (s5)	0	0x00000000	0b00000000000000000000000000000000
0	x22 (s6)	0	0x00000000	0b00000000000000000000000000000000
0	x23 (s7)	0	0x00000000	0b00000000000000000000000000000000
0	x24 (s8)	0	0x00000000	0b00000000000000000000000000000000
0	x25 (s9)	0	0x00000000	0b00000000000000000000000000000000
0	x26 (s10)	0	0x00000000	0b00000000000000000000000000000000
0	x27 (s11)	0	0x00000000	0b00000000000000000000000000000000
0	x28 (t3)	0	0x00000000	0b00000000000000000000000000000000
0	x29 (t4)	0	0x00000000	0b00000000000000000000000000000000
0	x30 (t5)	0	0x00000000	0b00000000000000000000000000000000
0	x31 (t6)	0	0x00000000	0b00000000000000000000000000000000

Memory Address 0x00000000

Memory Address	Decimal	Hex	Binary
0x00000000	0	0x00000000	0b00000000000000000000000000000000
0x00000004	5	0x00000005	0b00000000000000000000000000000101
0x00000008	8	0x00000008	0b00000000000000000000000000001000
0x0000000c	3	0x00000003	0b00000000000000000000000000000011
0x00000010	1	0x00000001	0b00000000000000000000000000000001
0x00000014	4	0x00000004	0b00000000000000000000000000000100
0x00000018	0	0x00000000	0b00000000000000000000000000000000
0x0000001c	0	0x00000000	0b00000000000000000000000000000000
0x00000020	0	0x00000000	0b00000000000000000000000000000000
0x00000024	0	0x00000000	0b00000000000000000000000000000000



RISC-V Interpreter

Input your RISC-V code here:

```
#creating array and saving it to memory (you can change the values manu
addi x1,x0,5
sw x1,4(x0)
addi x1,x0,8
sw x1,8(x0)
addi x1,x0,3
sw x1,12(x0)
addi x1,x0,1
sw x1,16(x0)
addi x1,x0,4
sw x1,20(x0)

#save number corresponding to total number of instances to x4
addi x4, x0, 5
#save number to x6 which will be deleted from x4 to check if all cells are sorted
addi x6, x0, 1
```

```
[line 43]: blt x1,x2,compare4
[line 50]: lw x1,16(x0)
[line 51]: lw x2,20(x0)
[line 52]: blt x1,x2,check
[line 61]: beq x4,x5,exit
No more instructions to run! Press Reset to reload the code!
```

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
0	x1 (ra)	5	0x00000005	0b00000000000000000000000000000101
0	x2 (sp)	8	0x00000008	0b00000000000000000000000000001000
0	x3 (gp)	0	0x00000000	0b00000000000000000000000000000000
0	x4 (tp)	5	0x00000005	0b00000000000000000000000000000101
0	x5 (t0)	5	0x00000005	0b00000000000000000000000000000101
0	x6 (t1)	1	0x00000001	0b00000000000000000000000000000001
0	x7 (t2)	0	0x00000000	0b00000000000000000000000000000000
0	x8 (s0/fp)	0	0x00000000	0b00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0b00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	0b00000000000000000000000000000000
0	x12 (a2)	0	0x00000000	0b00000000000000000000000000000000
0	x13 (a3)	0	0x00000000	0b00000000000000000000000000000000
0	x14 (a4)	0	0x00000000	0b00000000000000000000000000000000
0	x15 (a5)	0	0x00000000	0b00000000000000000000000000000000
0	x16 (a6)	0	0x00000000	0b00000000000000000000000000000000
0	x17 (a7)	0	0x00000000	0b00000000000000000000000000000000
0	x18 (s2)	0	0x00000000	0b00000000000000000000000000000000
0	x19 (s3)	0	0x00000000	0b00000000000000000000000000000000
0	x20 (s4)	0	0x00000000	0b00000000000000000000000000000000
0	x21 (s5)	0	0x00000000	0b00000000000000000000000000000000
0	x22 (s6)	0	0x00000000	0b00000000000000000000000000000000
0	x23 (s7)	0	0x00000000	0b00000000000000000000000000000000
0	x24 (s8)	0	0x00000000	0b00000000000000000000000000000000
0	x25 (s9)	0	0x00000000	0b00000000000000000000000000000000
0	x26 (s10)	0	0x00000000	0b00000000000000000000000000000000
0	x27 (s11)	0	0x00000000	0b00000000000000000000000000000000
0	x28 (t3)	0	0x00000000	0b00000000000000000000000000000000
0	x29 (t4)	0	0x00000000	0b00000000000000000000000000000000
0	x30 (t5)	0	0x00000000	0b00000000000000000000000000000000
0	x31 (t6)	0	0x00000000	0b00000000000000000000000000000000

Features

- *Reset* to load the code, *Step* one instruction, or *Run* all instructions
- Set a breakpoint by clicking on the line number (only for *Run*)
- View registers on the right, memory on the bottom of this page

Supported Instructions

- Arithmetics: ADD , ADDI , SUB
- Logical: AND , ANDI , OR , ORI , XOR , XORI
- Sets: SLT , SLTI , SLTU , SLTIU
- Shifts: SRA , SRAI , SRL , SRLI SLL , SLLI
- Memory: LW , SW , LB , SB
- PC: LUI , AUIPC
- Jumps: JAL , JALR
- Branches: BEQ , BNE , BLT , BGE , BLTU , BGEU

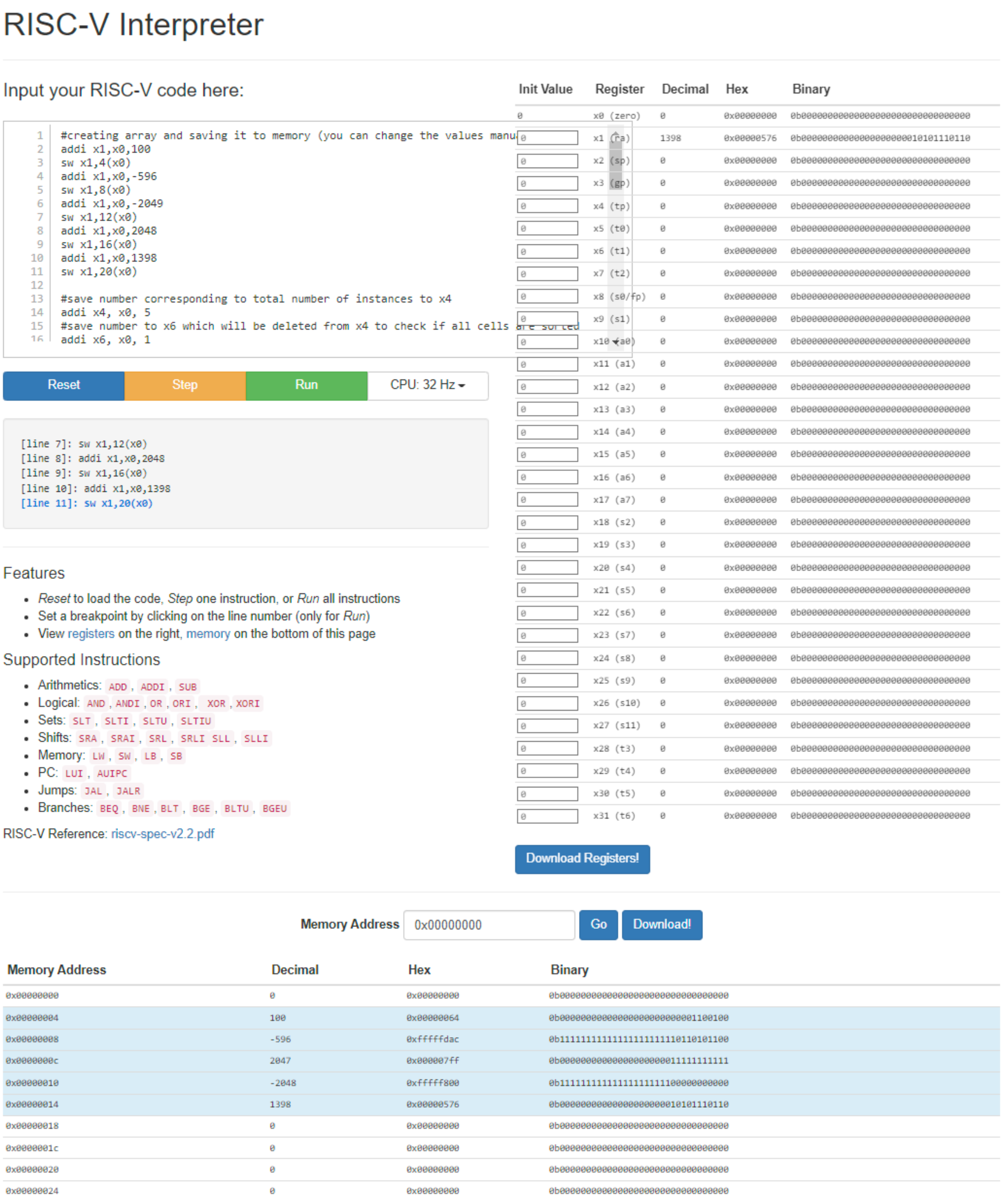
RISC-V Reference: riscv-spec-v2.2.pdf

Memory Address 0x00000000

Memory Address	Decimal	Hex	Binary
0x00000000	0	0x00000000	0b00000000000000000000000000000000
0x00000004	1	0x00000001	0b00000000000000000000000000000001
0x00000008	3	0x00000003	0b00000000000000000000000000000011
0x0000000c	4	0x00000004	0b00000000000000000000000000000100
0x00000010	5	0x00000005	0b00000000000000000000000000000101
0x00000014	8	0x00000008	0b00000000000000000000000000001000
0x00000018	0	0x00000000	0b00000000000000000000000000000000
0x0000001c	0	0x00000000	0b00000000000000000000000000000000
0x00000020	0	0x00000000	0b00000000000000000000000000000000
0x00000024	0	0x00000000	0b00000000000000000000000000000000

From the first test where array is set to [5,8,3,1,4] program sorts it correctly and exits when done.

Second test:



RISC-V Interpreter

Input your RISC-V code here:

```
#creating array and saving it to memory (you can change the values manu
addi x1,x0,100
sw x1,4(x0)
addi x1,x0,-596
sw x1,8(x0)
addi x1,x0,-2049
sw x1,12(x0)
addi x1,x0,2048
sw x1,16(x0)
addi x1,x0,1398
sw x1,20(x0)

#save number corresponding to total number of instances to x4
addi x4, x0, 5
#save number to x6 which will be deleted from x4 to check if all cells are sorted
addi x6, x0, 1
```

```
[line 7]: sw x1,12(x0)
[line 8]: addi x1,x0,2048
[line 9]: sw x1,16(x0)
[line 10]: addi x1,x0,1398
[line 11]: sw x1,20(x0)
```

Features

- *Reset* to load the code, *Step* one instruction, or *Run* all instructions
- Set a breakpoint by clicking on the line number (only for *Run*)
- View registers on the right, memory on the bottom of this page

Supported Instructions

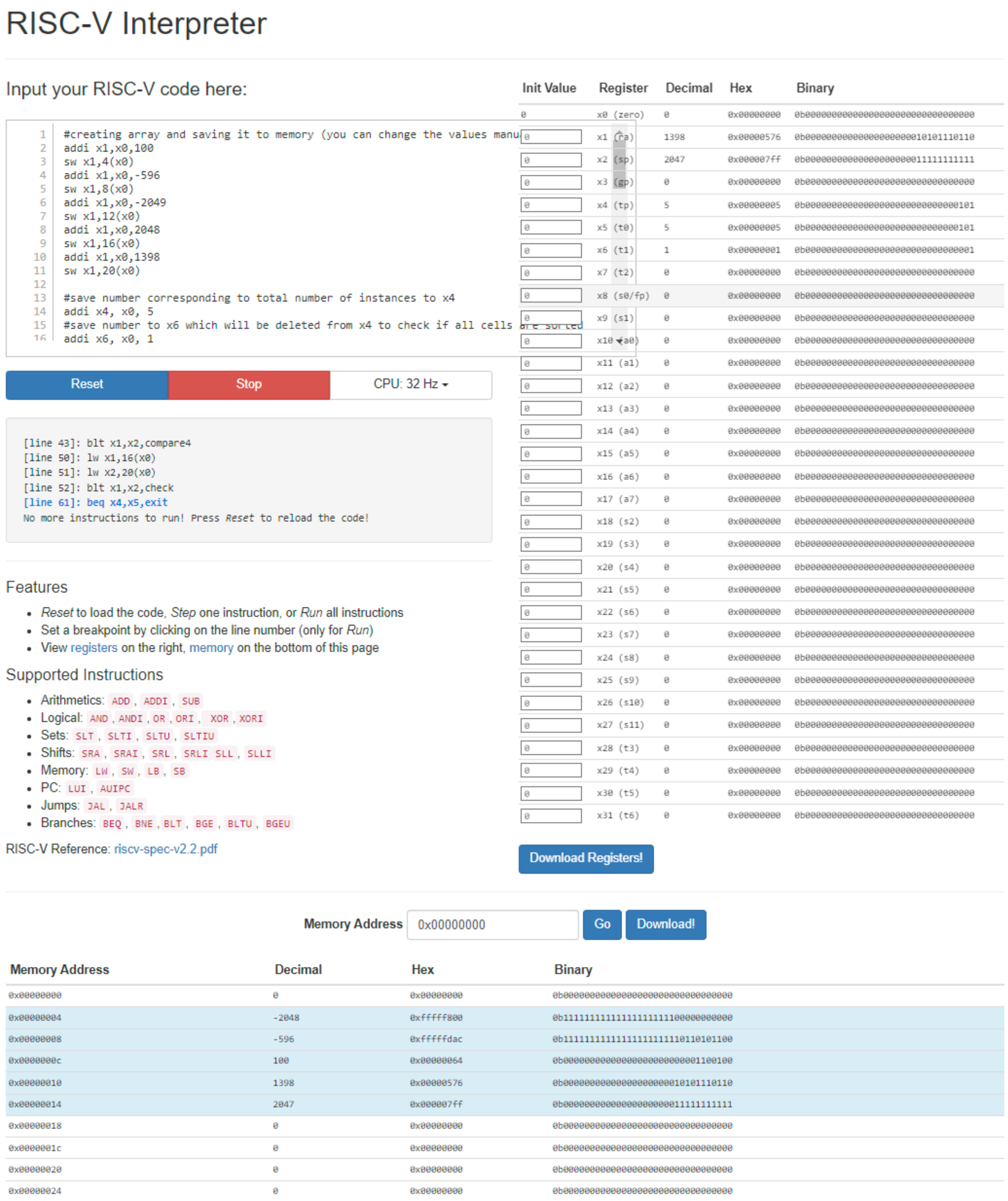
- Arithmetics: `ADD`, `ADDI`, `SUB`
- Logical: `AND`, `ANDI`, `OR`, `ORI`, `XOR`, `XORI`
- Sets: `SLT`, `SLTI`, `SLTU`, `SLTIU`
- Shifts: `SRA`, `SRAI`, `SRL`, `SRLI` `SLL`, `SLLI`
- Memory: `LW`, `SW`, `LB`, `SB`
- PC: `LUI`, `AUIPC`
- Jumps: `JAL`, `JALR`
- Branches: `BEQ`, `BNE`, `BLT`, `BGE`, `BLTU`, `BGEU`

RISC-V Reference: riscv-spec-v2.2.pdf

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
0	x1 (ra)	1398	0x00000576	0b00000000000000000000010101110110
0	x2 (sp)	0	0x00000000	0b00000000000000000000000000000000
0	x3 (gp)	0	0x00000000	0b00000000000000000000000000000000
0	x4 (tp)	0	0x00000000	0b00000000000000000000000000000000
0	x5 (t0)	0	0x00000000	0b00000000000000000000000000000000
0	x6 (t1)	0	0x00000000	0b00000000000000000000000000000000
0	x7 (t2)	0	0x00000000	0b00000000000000000000000000000000
0	x8 (s0/fp)	0	0x00000000	0b00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0b00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	0b00000000000000000000000000000000
0	x12 (a2)	0	0x00000000	0b00000000000000000000000000000000
0	x13 (a3)	0	0x00000000	0b00000000000000000000000000000000
0	x14 (a4)	0	0x00000000	0b00000000000000000000000000000000
0	x15 (a5)	0	0x00000000	0b00000000000000000000000000000000
0	x16 (a6)	0	0x00000000	0b00000000000000000000000000000000
0	x17 (a7)	0	0x00000000	0b00000000000000000000000000000000
0	x18 (s2)	0	0x00000000	0b00000000000000000000000000000000
0	x19 (s3)	0	0x00000000	0b00000000000000000000000000000000
0	x20 (s4)	0	0x00000000	0b00000000000000000000000000000000
0	x21 (s5)	0	0x00000000	0b00000000000000000000000000000000
0	x22 (s6)	0	0x00000000	0b00000000000000000000000000000000
0	x23 (s7)	0	0x00000000	0b00000000000000000000000000000000
0	x24 (s8)	0	0x00000000	0b00000000000000000000000000000000
0	x25 (s9)	0	0x00000000	0b00000000000000000000000000000000
0	x26 (s10)	0	0x00000000	0b00000000000000000000000000000000
0	x27 (s11)	0	0x00000000	0b00000000000000000000000000000000
0	x28 (t3)	0	0x00000000	0b00000000000000000000000000000000
0	x29 (t4)	0	0x00000000	0b00000000000000000000000000000000
0	x30 (t5)	0	0x00000000	0b00000000000000000000000000000000
0	x31 (t6)	0	0x00000000	0b00000000000000000000000000000000

Memory Address 0x00000000

Memory Address	Decimal	Hex	Binary
0x00000000	0	0x00000000	0b00000000000000000000000000000000
0x00000004	100	0x00000064	0b00000000000000000000000001100100
0x00000008	-596	0xfffffdac	0b11111111111111111111110110101100
0x0000000c	2047	0x000007ff	0b00000000000000000000011111111111
0x00000010	-2048	0xfffff800	0b11111111111111111111100000000000
0x00000014	1398	0x00000576	0b00000000000000000000010101110110
0x00000018	0	0x00000000	0b00000000000000000000000000000000
0x0000001c	0	0x00000000	0b00000000000000000000000000000000
0x00000020	0	0x00000000	0b00000000000000000000000000000000
0x00000024	0	0x00000000	0b00000000000000000000000000000000



RISC-V Interpreter

Input your RISC-V code here:

```
#creating array and saving it to memory (you can change the values manu
addi x1,x0,100
sw x1,4(x0)
addi x1,x0,-596
sw x1,8(x0)
addi x1,x0,-2049
sw x1,12(x0)
addi x1,x0,2048
sw x1,16(x0)
addi x1,x0,1398
sw x1,20(x0)

#save number corresponding to total number of instances to x4
addi x4, x0, 5
#save number to x6 which will be deleted from x4 to check if all cells are sorted
addi x6, x0, 1
```

```
[line 43]: blt x1,x2,compare4
[line 50]: lw x1,16(x0)
[line 51]: lw x2,20(x0)
[line 52]: blt x1,x2,check
[line 61]: beq x4,x5,exit
No more instructions to run! Press Reset to reload the code!
```

Features

- *Reset* to load the code, *Step* one instruction, or *Run* all instructions
- Set a breakpoint by clicking on the line number (only for *Run*)
- View registers on the right, memory on the bottom of this page

Supported Instructions

- Arithmetics: `ADD` , `ADDI` , `SUB`
- Logical: `AND` , `ANDI` , `OR` , `ORI` , `XOR` , `XORI`
- Sets: `SLT` , `SLTI` , `SLTU` , `SLTIU`
- Shifts: `SRA` , `SRAI` , `SRL` , `SRLI` `SLL` , `SLLI`
- Memory: `LW` , `SW` , `LB` , `SB`
- PC: `LUI` , `AUIPC`
- Jumps: `JAL` , `JALR`
- Branches: `BEQ` , `BNE` , `BLT` , `BGE` , `BLTU` , `BGEU`

RISC-V Reference: riscv-spec-v2.2.pdf

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
0	x1 (ra)	1398	0x00000576	0b00000000000000000000010101110110
0	x2 (sp)	2047	0x000007ff	0b00000000000000000000011111111111
0	x3 (gp)	0	0x00000000	0b00000000000000000000000000000000
0	x4 (tp)	5	0x00000005	0b00000000000000000000000000000101
0	x5 (t0)	5	0x00000005	0b00000000000000000000000000000101
0	x6 (t1)	1	0x00000001	0b00000000000000000000000000000001
0	x7 (t2)	0	0x00000000	0b00000000000000000000000000000000
0	x8 (s0/fp)	0	0x00000000	0b00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0b00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	0b00000000000000000000000000000000
0	x12 (a2)	0	0x00000000	0b00000000000000000000000000000000
0	x13 (a3)	0	0x00000000	0b00000000000000000000000000000000
0	x14 (a4)	0	0x00000000	0b00000000000000000000000000000000
0	x15 (a5)	0	0x00000000	0b00000000000000000000000000000000
0	x16 (a6)	0	0x00000000	0b00000000000000000000000000000000
0	x17 (a7)	0	0x00000000	0b00000000000000000000000000000000
0	x18 (s2)	0	0x00000000	0b00000000000000000000000000000000
0	x19 (s3)	0	0x00000000	0b00000000000000000000000000000000
0	x20 (s4)	0	0x00000000	0b00000000000000000000000000000000
0	x21 (s5)	0	0x00000000	0b00000000000000000000000000000000
0	x22 (s6)	0	0x00000000	0b00000000000000000000000000000000
0	x23 (s7)	0	0x00000000	0b00000000000000000000000000000000
0	x24 (s8)	0	0x00000000	0b00000000000000000000000000000000
0	x25 (s9)	0	0x00000000	0b00000000000000000000000000000000
0	x26 (s10)	0	0x00000000	0b00000000000000000000000000000000
0	x27 (s11)	0	0x00000000	0b00000000000000000000000000000000
0	x28 (t3)	0	0x00000000	0b00000000000000000000000000000000
0	x29 (t4)	0	0x00000000	0b00000000000000000000000000000000
0	x30 (t5)	0	0x00000000	0b00000000000000000000000000000000
0	x31 (t6)	0	0x00000000	0b00000000000000000000000000000000

Memory Address: 0x00000000

Memory Address	Decimal	Hex	Binary
0x00000000	0	0x00000000	0b00000000000000000000000000000000
0x00000004	-2048	0xfffff800	0b11111111111111111111100000000000
0x00000008	-596	0xfffffdac	0b11111111111111111111110110101100
0x0000000c	100	0x00000064	0b00000000000000000000000001100100
0x00000010	1398	0x00000576	0b00000000000000000000010101110110
0x00000014	2047	0x000007ff	0b00000000000000000000011111111111
0x00000018	0	0x00000000	0b00000000000000000000000000000000
0x0000001c	0	0x00000000	0b00000000000000000000000000000000
0x00000020	0	0x00000000	0b00000000000000000000000000000000
0x00000024	0	0x00000000	0b00000000000000000000000000000000

From the second test I find out that if I set number to 2048 =< x <= 2049 program does not work, because it does not understand the values greater than 2047 or lower than -2048.

5. Inference

From the testing I found out that the algorithm works if any of the numbers are not equal with each other, if the numbers are greater than 2047 or less than -2048 it does not work. That is because addi’s immediate field is limited to [-2048, 2047]. Algorithm is also quite limited because it does not support more than five at time, at least in this version.