ACA Submission 1:

Scalar Processor Simulator

Tymoteusz Suszczynski

Processor Architecture

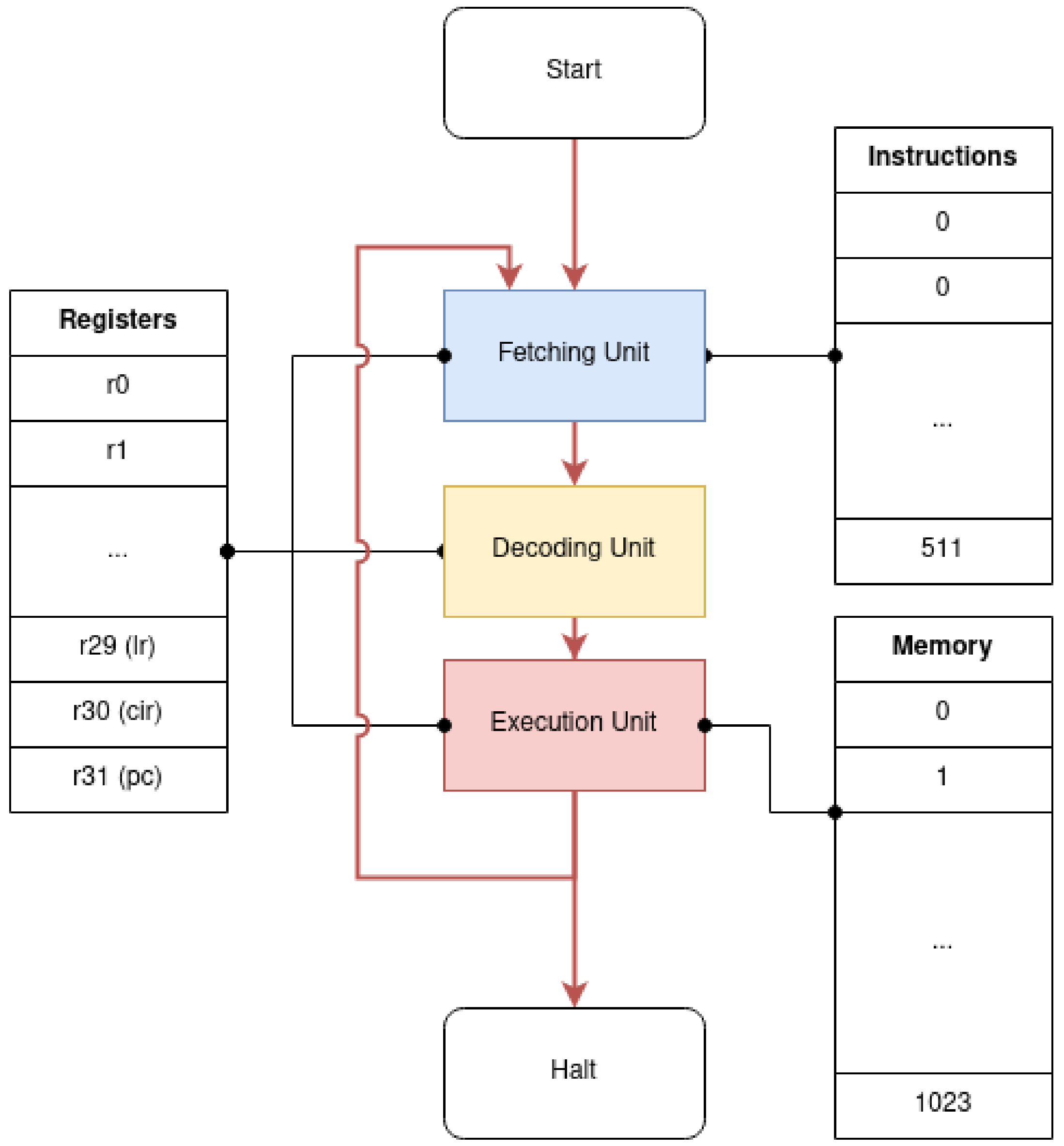
The design follows a Harvard architecture and consists of 3 units for fetching, decoding and executing. The processor is scalar, so evaluating each of the units takes one cycle, meaning 3 cycles are spent per instruction.

The fetching unit takes the instruction in INSTR[pc] and stores it in REG[30] (cir).

The decoding unit takes the instruction in REG[30], decodes the instruction and passes the result to the execution unit.

The execution unit carries out the instruction. It is able to manipulate the register file as well as memory.

This cycle continues until the halt instruction is reached and execution stops.



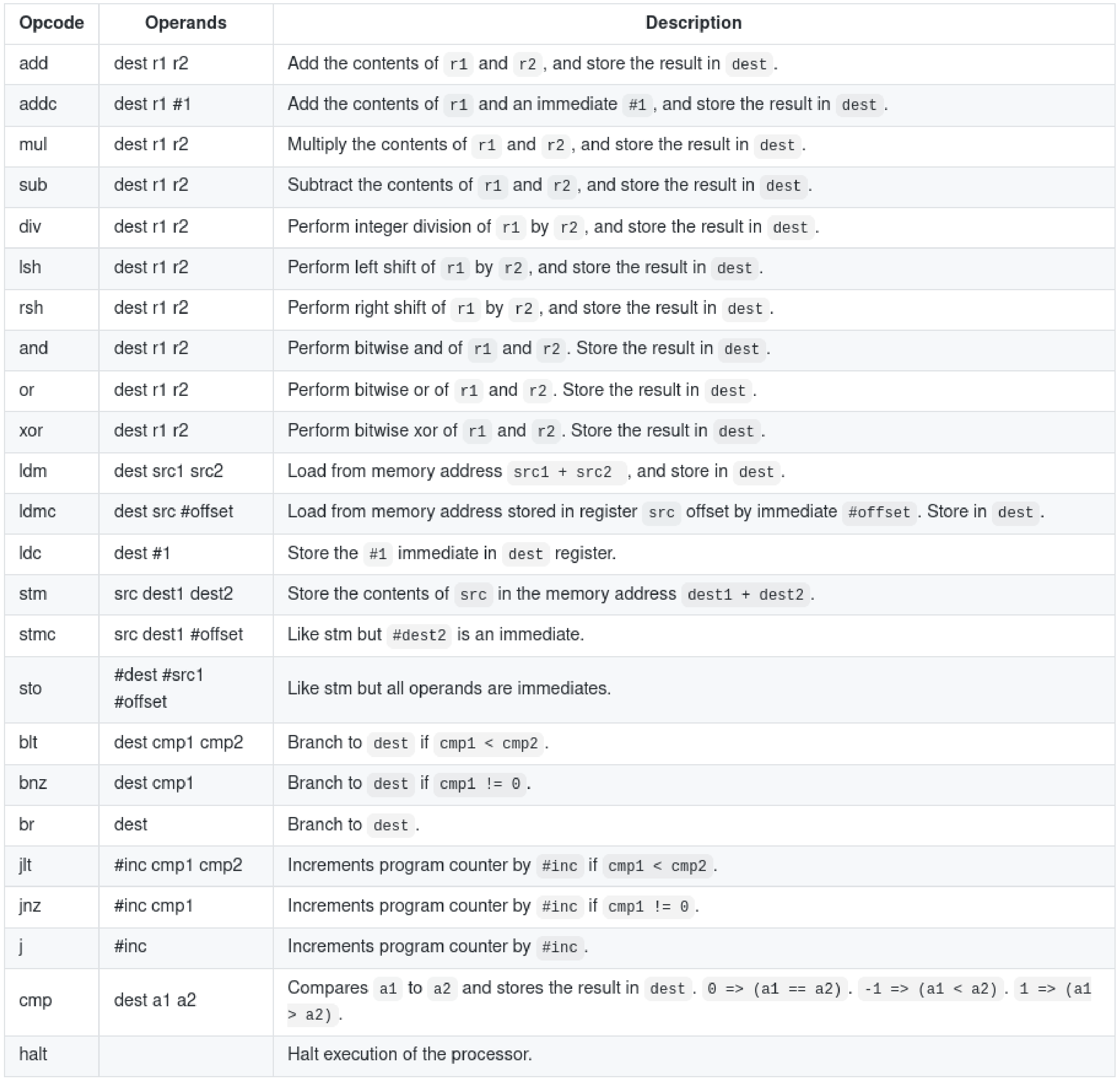
Instruction Set Architecture

The following table describes the supported instructions and their function.

Instructions have been chosen in order to make interesting programs comfortable enough to write, while also keeping the process of assembling simple and being realistic with regards to the architecture.

For example, most instructions use register addressing modes, with the exception of a few which may use offset or immediate addressing. This is due to instructions requiring additional versions of themselves internally to support various addressing modes.

Therefore, I have added instructions which support this only where I believe it is useful for implementing structures such as loops, counters and accessing higher addresses in memory.



Opcode	Operands	Description
add	dest r1 r2	Add the contents of `r1` and `r2`, and store the result in `dest`.
addc	dest r1 #1	Add the contents of `r1` and an immediate `#1`, and store the result in `dest`.
mul	dest r1 r2	Multiply the contents of `r1` and `r2`, and store the result in `dest`.
sub	dest r1 r2	Subtract the contents of `r1` and `r2`, and store the result in `dest`.
div	dest r1 r2	Perform integer division of `r1` by `r2`, and store the result in `dest`.
lsh	dest r1 r2	Perform left shift of `r1` by `r2`, and store the result in `dest`.
rsh	dest r1 r2	Perform right shift of `r1` by `r2`, and store the result in `dest`.
and	dest r1 r2	Perform bitwise and of `r1` and `r2`. Store the result in `dest`.
or	dest r1 r2	Perform bitwise or of `r1` and `r2`. Store the result in `dest`.
xor	dest r1 r2	Perform bitwise xor of `r1` and `r2`. Store the result in `dest`.
ldm	dest src1 src2	Load from memory address `src1 + src2`, and store in `dest`.
ldmc	dest src #offset	Load from memory address stored in register `src` offset by immediate `#offset`. Store in `dest`.
ldc	dest #1	Store the `#1` immediate in `dest` register.
stm	src dest1 dest2	Store the contents of `src` in the memory address `dest1 + dest2`.
stmc	src dest1 #offset	Like stm but `#dest2` is an immediate.
sto	#dest #src1 #offset	Like stm but all operands are immediates.
blt	dest cmp1 cmp2	Branch to `dest` if `cmp1 < cmp2`.
bnz	dest cmp1	Branch to `dest` if `cmp1 != 0`.
br	dest	Branch to `dest`.
jlt	#inc cmp1 cmp2	Increments program counter by `#inc` if `cmp1 < cmp2`.
jnz	#inc cmp1	Increments program counter by `#inc` if `cmp1 != 0`.
j	#inc	Increments program counter by `#inc`.
cmp	dest a1 a2	Compares `a1` to `a2` and stores the result in `dest`. `0 => (a1 == a2)`. `-1 => (a1 < a2)`. `1 => (a1 > a2)`.
halt		Halt execution of the processor.

Language Choice

I have made C++ my language of choice in writing this simulator. The main reasons for this choice are:

- Excellent control with regards to managing memory. This is useful when simulating registers, main memory, and instruction memory.
- Object-oriented functionality will allow my simulator to be well structured and easily extensible in the future.
- (mainly) I am comfortable programming in the language.

Running programs & usage

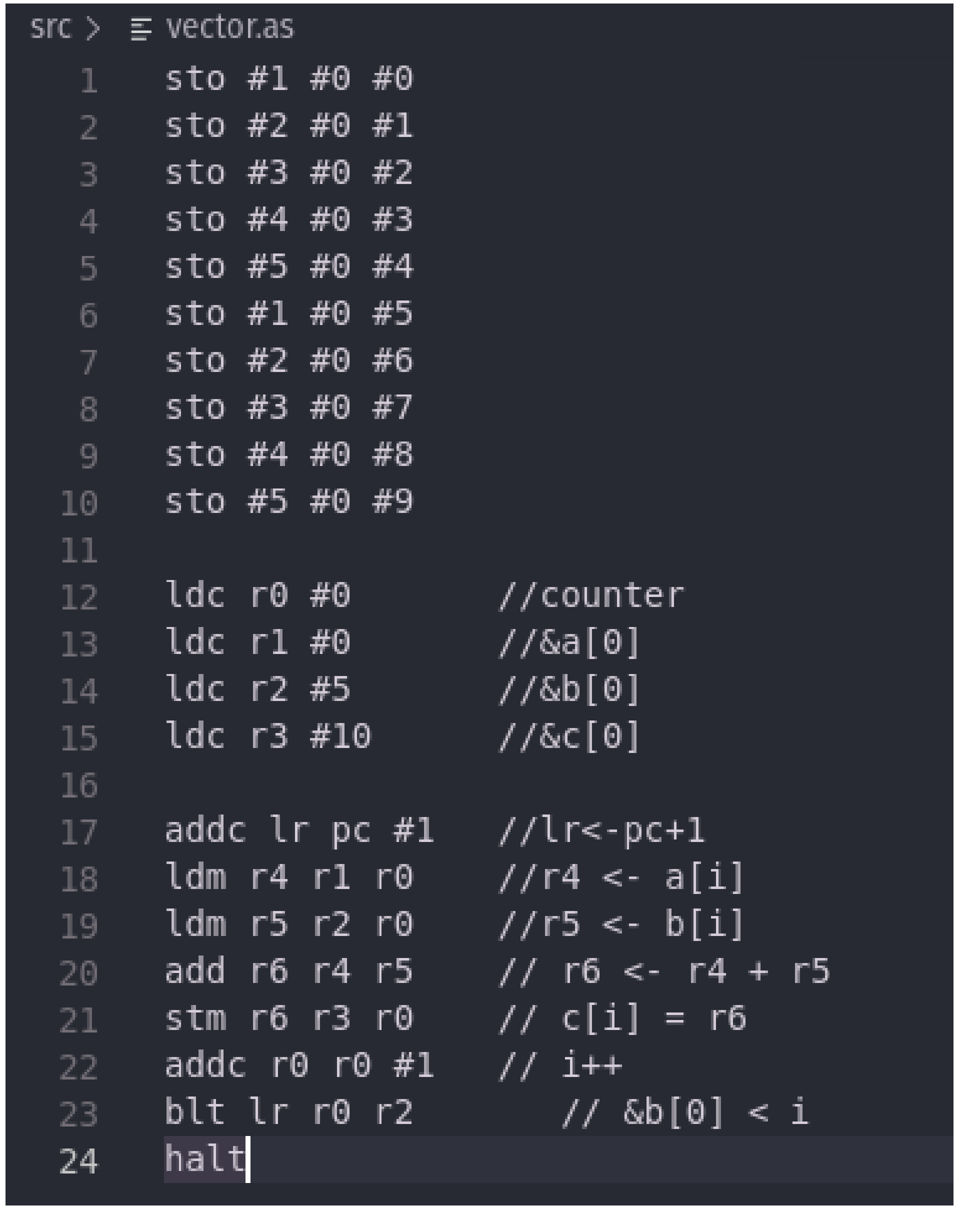
- The `assembler` and `proc` programs may be compiled using the Makefile:
  - `make proc`
  - `make assembler`
- The submission contiains the assembly and binary code for 3 benchmarking programs:
  - `gcm.as`
  - `vector.as`
  - `bubble.as`
- Valid assembly files may be assembled to binary files with the `assembler` program included. For example, to assemble `gcm.as:`
  - `./assembler gcm.as gcm.o`
- These programs may then be executed using the simulator. For example, to execute the `gcm.o` program:
  - `./proc gcm.o`

Benchmarking: Vector addition

This vector addition benchmark initialises two vectors in memory and adds them. The resulting vector is stored in memory, starting at location 10.

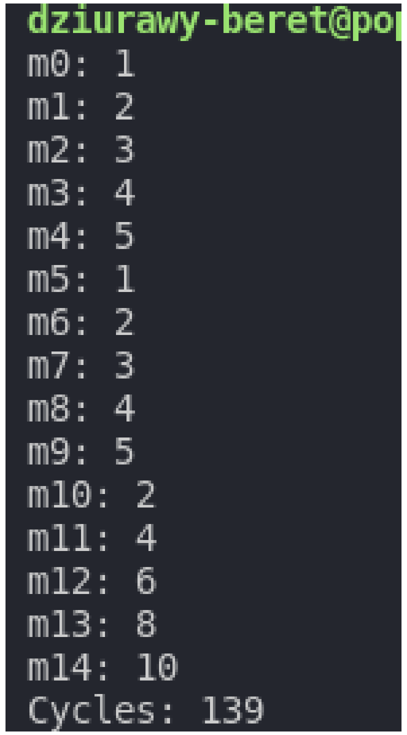
The image shows the output of the simulator upon running the program. The vector from m10 to m14 is a sum of the vectors from m0 to m4 and m5 to m9.

This benchmark completed in 139 cycles.



src > vector.as

```
sto #1 #0 #0
sto #2 #0 #1
sto #3 #0 #2
sto #4 #0 #3
sto #5 #0 #4
sto #1 #0 #5
sto #2 #0 #6
sto #3 #0 #7
sto #4 #0 #8
sto #5 #0 #9

ldc r0 #0       //counter
ldc r1 #0       //&a[0]
ldc r2 #5       //&b[0]
ldc r3 #10      //&c[0]

addc lr pc #1   //lr<-pc+1
ldm r4 r1 r0    //r4 <- a[i]
ldm r5 r2 r0    //r5 <- b[i]
add r6 r4 r5    // r6 <- r4 + r5
stm r6 r3 r0    // c[i] = r6
addc r0 r0 #1   // i++
blt lr r0 r2       // &b[0] < i
halt
```



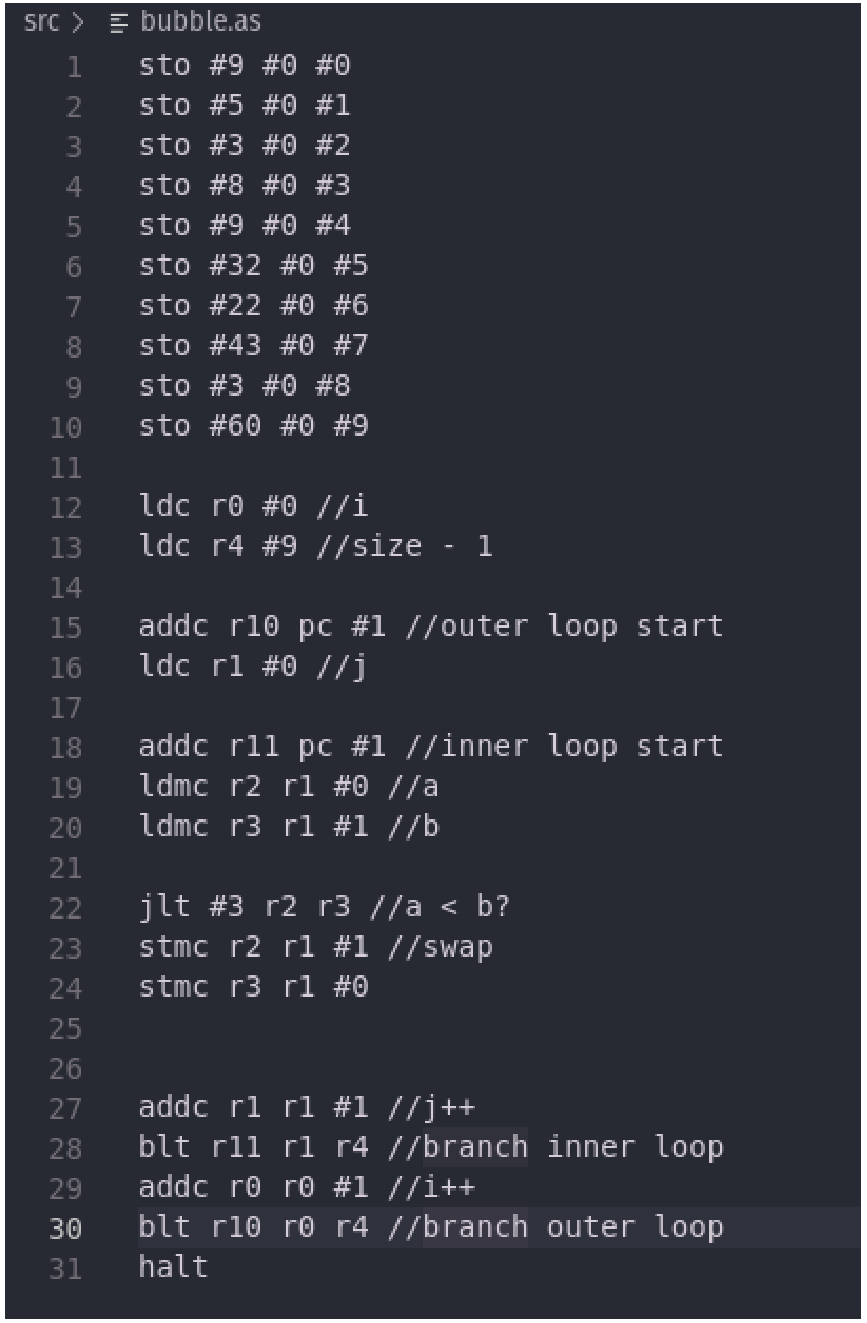
```
dziurawy-beret@po
m0: 1
m1: 2
m2: 3
m3: 4
m4: 5
m5: 1
m6: 2
m7: 3
m8: 4
m9: 5
m10: 2
m11: 4
m12: 6
m13: 8
m14: 10
Cycles: 139
```

Benchmarking: Bubble sort

•This bubble sort benchmark initialises an array {9, 5, 3, 8, 9, 32, 22, 43, 3, 60} and then sorts it using the bubble sort algorithm.

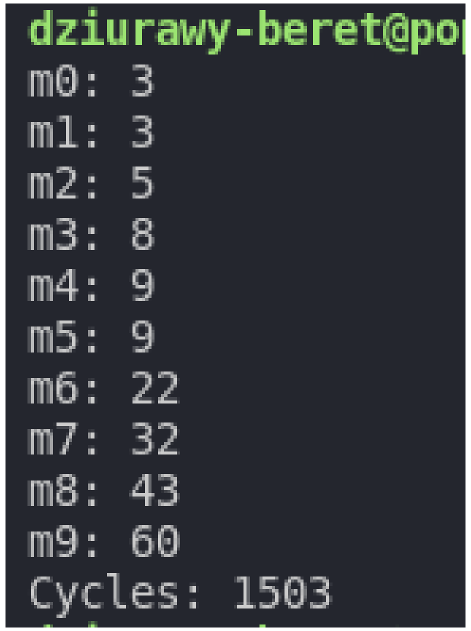
•The image shows the output of the simulator upon running the program. Only non-zero memory locations are shown. As we can see, the list is sorted.

•This benchmark completed in 1503 cycles.



src > bubble.as

```
sto #9 #0 #0
sto #5 #0 #1
sto #3 #0 #2
sto #8 #0 #3
sto #9 #0 #4
sto #32 #0 #5
sto #22 #0 #6
sto #43 #0 #7
sto #3 #0 #8
sto #60 #0 #9

ldc r0 #0 //i
ldc r4 #9 //size - 1

addc r10 pc #1 //outer loop start
ldc r1 #0 //j

addc r11 pc #1 //inner loop start
ldmc r2 r1 #0 //a
ldmc r3 r1 #1 //b

jlt #3 r2 r3 //a < b?
stmc r2 r1 #1 //swap
stmc r3 r1 #0


addc r1 r1 #1 //j++
blt r11 r1 r4 //branch inner loop
addc r0 r0 #1 //i++
blt r10 r0 r4 //branch outer loop
halt
```

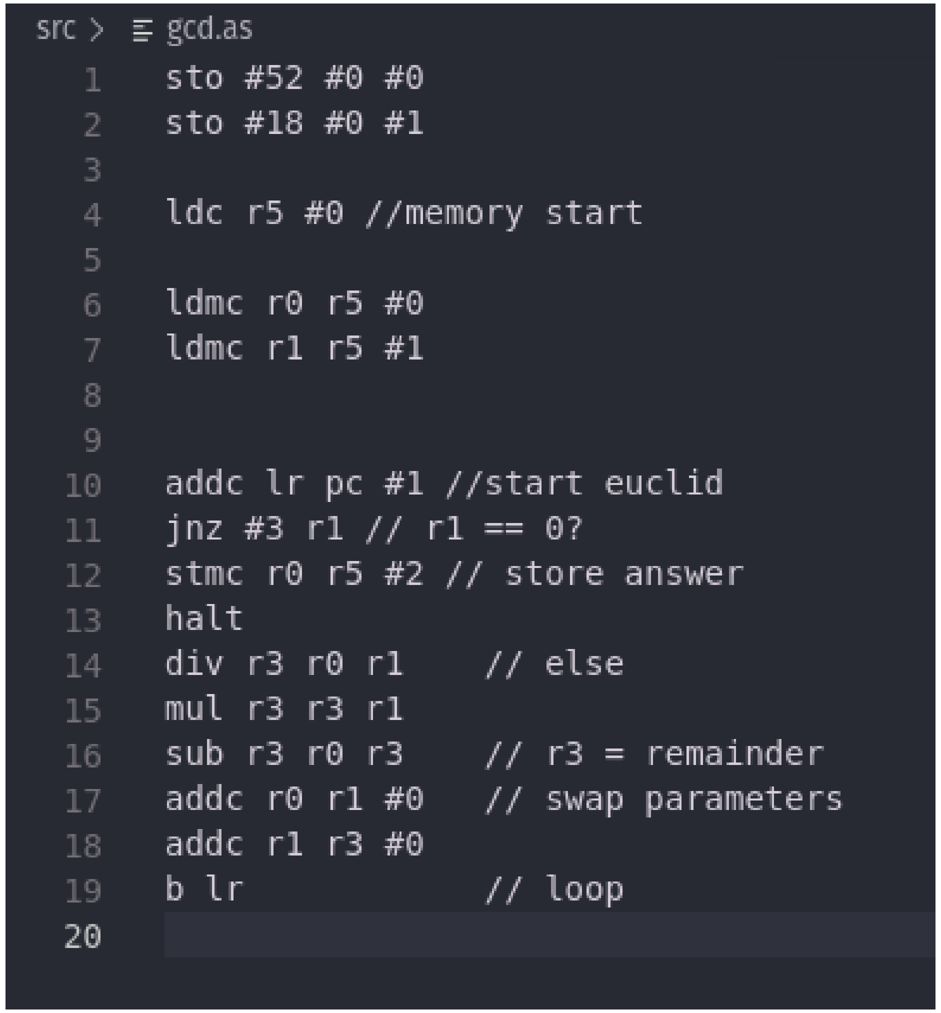


```
dziurawy-beret@po
m0: 3
m1: 3
m2: 5
m3: 8
m4: 9
m5: 9
m6: 22
m7: 32
m8: 43
m9: 60
Cycles: 1503
```

Benchmarking: Euclidean algorithm

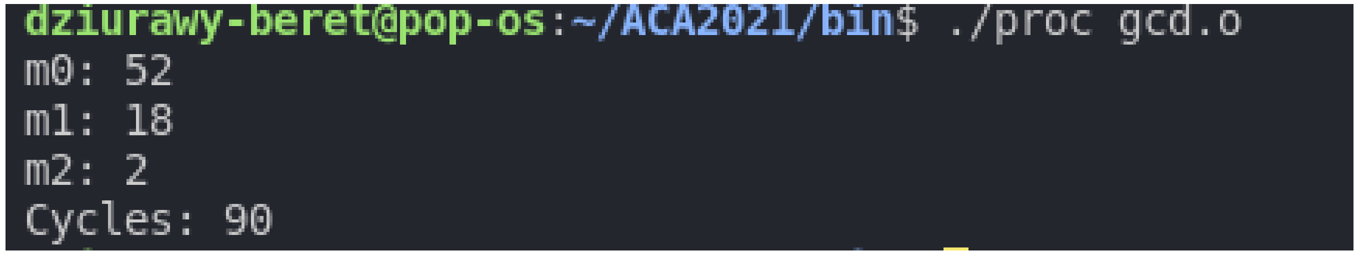
•This benchmark calculated the greatest common divisor between two numbers.
•The output on the right shows m0 and m1 containing the two numbers upon which the algorithm is carried out, as well as m2 containing the result.

•This benchmark completed in 90 cycles.



src > gcd.as

```
sto #52 #0 #0
sto #18 #0 #1

ldc r5 #0 //memory start

ldmc r0 r5 #0
ldmc r1 r5 #1


addc lr pc #1 //start euclid
jnz #3 r1 // r1 == 0?
stmc r0 r5 #2 // store answer
halt
div r3 r0 r1    // else
mul r3 r3 r1
sub r3 r0 r3    // r3 = remainder
addc r0 r1 #0   // swap parameters
addc r1 r3 #0
b lr            // loop

```



```
dziurawy-beret@pop-os:~/ACA2021/bin$ ./proc gcd.o
m0: 52
m1: 18
m2: 2
Cycles: 90
```