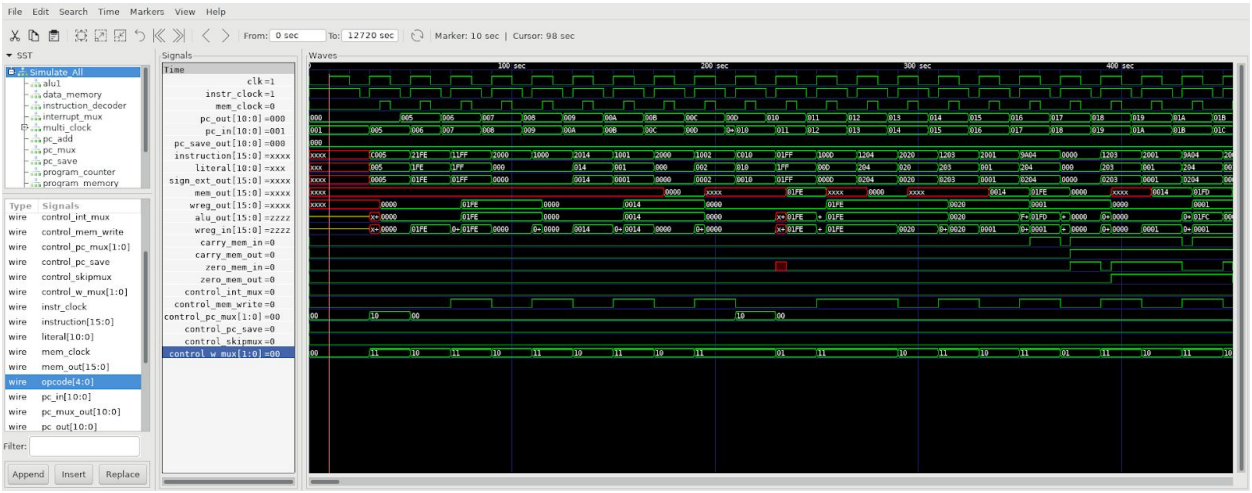
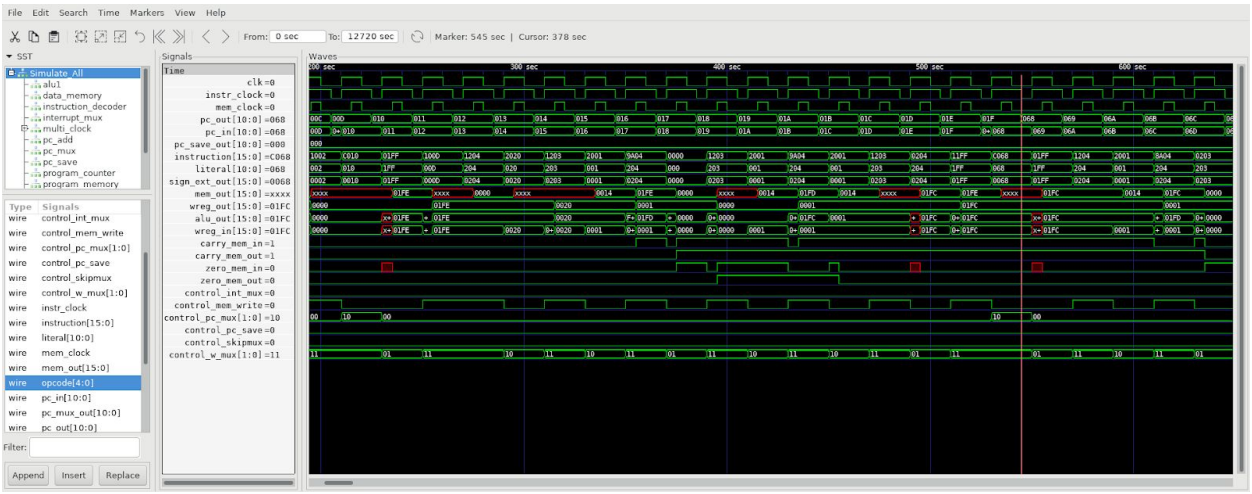
This document provides evidence that the ISA is functional. This document is best consumed with the program.mem file open side-by-side with this one. This file contains the C and assembly code as comments over the machine language instructions. These instructions are loaded into the program memory module by the simulator. Note that the waveform viewer defaults to 1 second per increment if the units are not specified. The processor is not intended to be run at 20 seconds per clock cycle.

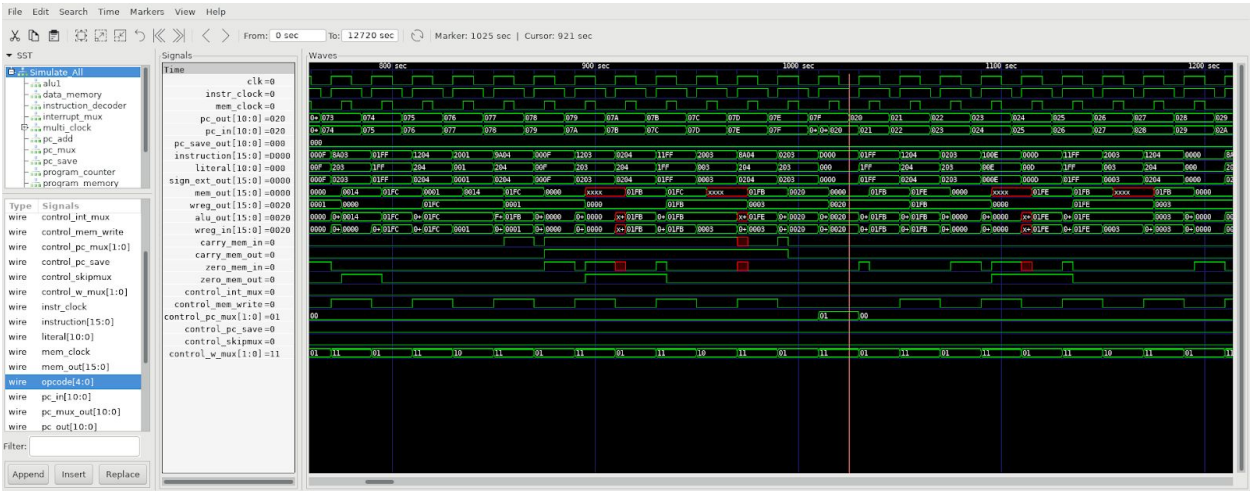
The position of the waveforms is consistent throughout these images. The base clock is labeled clk, which is split into an instr_clock and mem_clock which drive the program memory and data memory/working register respectively. Pc_out[10:0] is the 11 bit output of the program counter and is used to index the program memory which returns the value shown as instruction[15:0]. The pc_in[10:0] value shows the output of the multiplexers which set the program counter on the next negative edge of the instruction clock. The literal[10:0] value is the literal portion of the instruction[15:0] value. sign_ext_out[15:0] is the 16bit sign-extended version of the literal value. Mem_out, wreg_out, and alu_out are the outputs of the data memory, working register, and alu, respectively. Since the input to the working register is multiplexed, the input into the working register (wreg_in[15:0]) is shown. This value will be loaded into the working register on the next negative edge of the memory clock and will be output from the working register on the next positive edge of the memory clock. The carry and zero register values are shown with a direction with respect to the data memory module (e.g. carry_mem_in is output from the ALU and is an input to the data memory). Finally, the control signals are shown (all values with the “control_” prefix). These values are output from the instruction decoder.



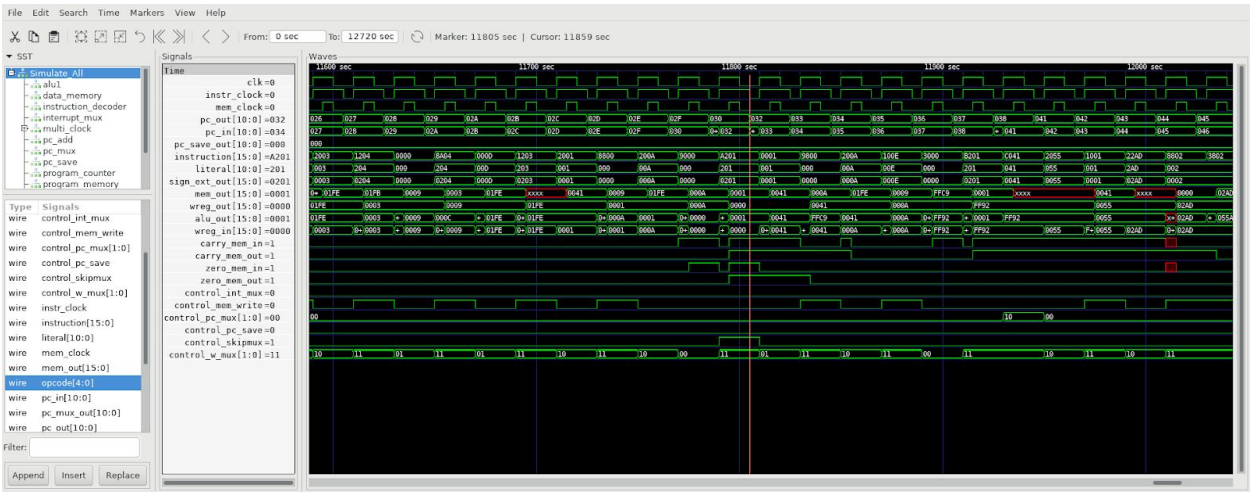
This image shows ISA startup. Instruction execution begins on the first falling edge of the instruction clock, which loads the first instruction, a jump to 0x005, the startup routine. The startup initializes the stack pointer and global variables and then jumps to the main routine (at 0x00A).



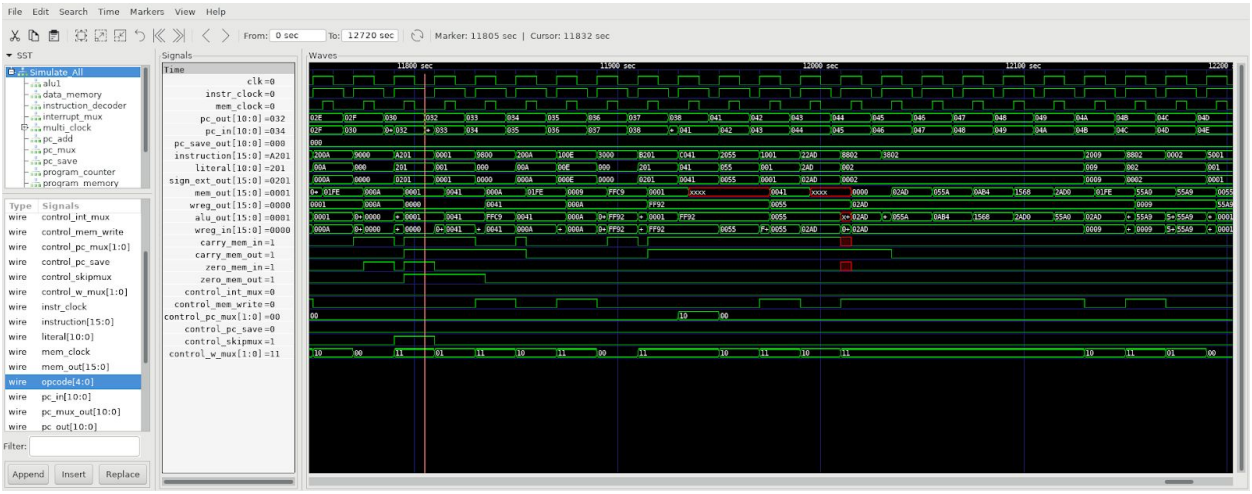
This image shows the rest of the for loop in the main routine, loads the function arguments and return pointer onto the state, and calls the add function by jumping to 0x068 (where the add routine is stored). The red marker is placed between the end of the jump (gol instruction) and the beginning of the first instruction of the add routine.



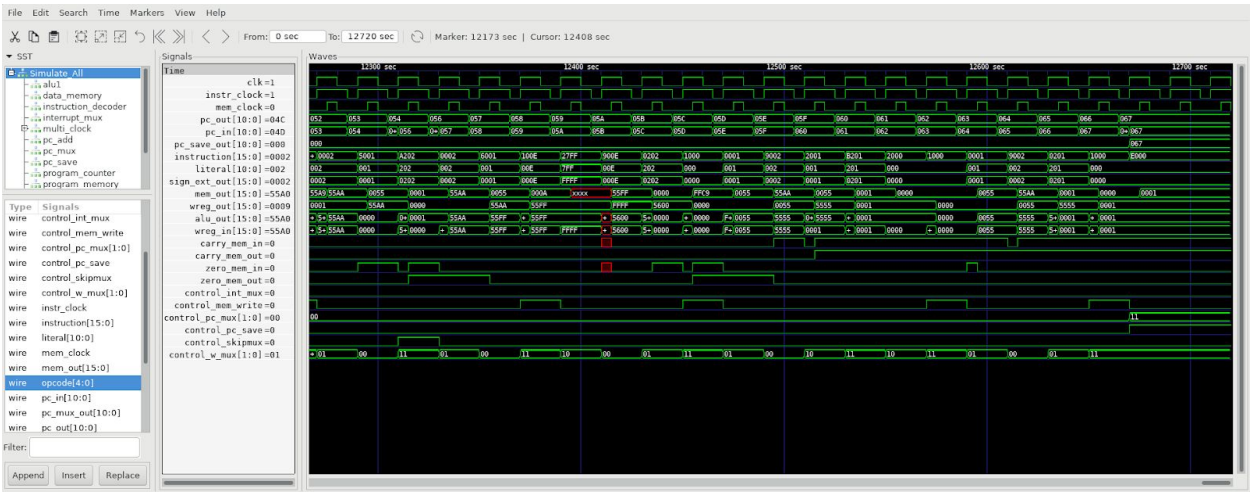
This image shows the add routine loading the return value onto the stack and jumping back to the for loop at 0x020. The calling function handles unloading the return value and restoring the stack. The red marker line is placed between the jump back to the main routine and the first instruction in the main routine. Note that the stack is “popped” by restoring the stackptr value to the same value before the call. This assumes that there is only one thread of execution, which is valid because an interrupt controller has not been implemented. If multiple threads were used, the calling function would have to increment the stack pointer by the appropriate amount (number of arguments + 1 for return value + 1 for program return address).



This image shows the exit of the for loop. The i variable is incremented, and is found to equal 10 by subtracting 10 from the i value. The sms ZERO (Skip if Memory value is Set) instruction skips over the gol MAIN instruction because the ZERO register (zero_mem_in/out) is set, exiting the for loop.



This image shows the i = i-j assignment, the if (i >= 10) j=0xAA else j = 0x55 statement, the k = 0x55AA assignment, and the beginnings of the while loop at PC = 0x04B. Note that the if statement (0x034 - 0x042) assumes that “i” could be negative and tests for that by rotating the i value left and checking to see what the value of the sign bit is. The only possibilities are mixed signs or two positive signs since the 10 value is a compile-time literal and therefore, could not be negative. More information about comparing mixed sign values can be found in the report under the Registers subsection of the Architecture section.



This portion shows the end of the while loop (which operates once before exiting, although the implementation used does 1 pass without jumping, 0 or 2+ operations would jump) and remaining comparisons. The wfi (Wait For Interrupt, the HALT equivalent instruction) executes at 0x067. The image shows that the program counter does not increment during any of the following cycles, and would not unless the interrupt port on the instruction decoder is pulled high by an interrupt controller. The interrupt controller was not implemented because it was out of scope for this project. The interrupt port is effectively hardwired to never trigger an interrupt.