OPERATING SYSTEMS

SIMULATOR OF A MULTIPROGRAMMED COMPUTER SYSTEM V1 (and following ones)

CHECKING ASSERTIONS

Introduction

To simplify testing the simulator in different situations, an assertion system has been built, allowing the user to check values stored in several simulator elements at different times.

This way, the user can write an *asserts* file containing the expected values for different elements at particular times. At those instants, expected values will be compared with the obtained ones, showing an error in case they differ.

Simulator modifications

- Added variable GEN_ASSERTS. It works as a flag, indicating whether the *asserts* file is used to generate an assertions file for the specified variables at the described times, or assertions should be checked. If GEN_ASSERTS equals 1, an assertions file is generated; if 0, assertions are checked. It is set to 1 if `--generateAsserts` is specified when invoking the simulator.
- The Clock (`Clock.c` and `Clock.h`) is used to know the corresponding time for each assertion.
- `Asserts.c` and `Asserts.h`. Contain the logic to verify assertions.
- `Wrappers.c`. Includes the call to `Assert_CheckAsserts` after the execution of `Processor_DecodeAndExecuteInstruction`.

```
void __wrap_Processor_DecodeAndExecuteInstruction() {
    __real_Processor_DecodeAndExecuteInstruction();
    Asserts_CheckAsserts();
}
```

  And the call to `Assert_ TerminateAssertions()` after the execution of `Processor_ InstructionCycleloop()`.

```
void __wrap_Processor_InstructionCycleLoop() {
  __real_Processor_InstructionCycleLoop();
  Asserts_TerminateAssertions();
```

- `asserts`: or the file specified with option `--assertsFile`, contains the assertions to be checked during the simulator execution. If it does not exist, nothing is checked (but a message is displayed to inform about it).

Overall functioning

Assertions are loaded into memory when the simulation starts.

If an assertions file (`asserts`) does not exist, assertions will not be checked.

Just after the execution of `Processor_DecodeAndExecuteInstruction`, whether an assertion exists for that time unit or not is checked. If it exists, the expected value is compared with the current one for the specified element. If they are different, a message is displayed.

Messages are always shown with `ComputerSystem_DebugMessage` in the ERROR section, so they are always displayed.

It is recommended to execute the simulator with debugging messages off (option "n") when checking assertions.

If `--generateAsserts` option is specified when invoking the simulator, the necessary lines to be copied inside an assertions file (`asserts`) will appear on the screen, instead of being checked during execution.

Assertions

Assertions must be stored in a file named `asserts`. Assertions have the following structure:

```
time, element, value [,address]
```

That line can be read as following: “after the specified **`time`**, the indicated **`element`** will contain **`value`**”. In case the element was a main memory position, the fourth value above will specify its **`address`**.

The `value` can be a string (for checking an operation code) or a numeric value, depending on the element to be checked.

Elements that can be checked are the following (can be consulted in `Asserts.h`):

- `RMEM_OP`: Operation code stored in the specified relative memory position of the executing process:
  `5, RMEM_OP, ADD, 3` (At time 5, the operation code in relative address 3 is ADD_INST).
- `RMEM_O1`: Operand 1 stored in the specified relative memory position of the executing process:
  `5, RMEM_O1, 132, 3` (At time 5, operand 1 in relative address 3 is 132).
- `RMEM_O2`: Operand 2 stored in the specified relative memory position of the executing process:
  `5, RMEM_O2, 13, 3` (At time 5, operand 2 in relative address 3 is 13).
- `AMEM_OP`: Operation code stored in the specified absolute memory position:
  `5, AMEM_OP, ADD, 140` (At time 5, the operation code in absolute address 140 is ADD_INST).
- `AMEM_O1`: Operand 1 stored in the specified absolute memory position:
  `5, AMEM_O1, 132, 140` (At time 5, operand 1 in absolute address 140 is 132).
- `AMEM_O2`: Operand 2 stored in the specified absolute memory position:
  `5, AMEM_O2, 13, 140` (At time 5, operand 2 in absolute address 140 is 13).
- `PC`: Contents of the program counter:
  `6, PC, 15` (At time 6, program counter contains 15).
- `ACC:` Contents of the accumulator:
  `3, ACC, 10` (At time 3, the accumulator contains 10).
- `IR_OP:` Instruction register’s operation code.
  `10, IR_OP, JUMP` (At time 10, the instruction register contains instruction JUMP_INST).
- `IR_O1:` Instruction register’s operand 1.
  `10, IR_O1, 2` (At time 10, the instruction register’s operand 1 contains 2).
- `IR_O2:` Instruction register’s operand 2.
  `10, IR_O2, 0` (At time 10, the instruction register’s operand 2 contains 0).
- `PSW:` Contents of the PSW register.
  `16, PSW, 1` (At time 16, the PSW register contains base-10 value 1).
- `MAR:` Contents of the MAR register.
  `20, MAR, 140` (At time 20, the MAR register contains 140).
- `MBR_OP:` MBR register’s operation code.
  `6, MBR_OP, ZJUMP` (At time 6, the MBR register ‘s operation code contains ZJUMP_INST).
- `MBR_O1:` MBR register’s operand 1.
  `6, MBR_01, 150` (At time 6, the MBR register’s operand 1 contains 150).
- `MBR_O2:` MBR register’s operand 2.
  `6, MBR_02, 10` (At time 6, the MBR register’s operand 2 contains 10).
- `MMU_BS:` Contents of the base register of the MMU.
  `6, MMU_BS, 60` (At time 6, the base register of the MMU contains 60).
- `MMU_LM:` Contents of the limit register of the MMU.
  `6, MMU_LM, 20` (At time 6, the limit register of the MMU contains 20).

- `MMU_MAR:` Contents of the MAR register of the MMU.

  `6, MMU_MAR, 7` (At time 6, the MAR register of the MMU contains 7).

- `MMEM_MAR:` Contents of the MAR register of the MainMemory.

  `6, MMEM_MAR, 15` (At time 6, the MAR register of the MainMemory contains 15).

- `MMBR_OP:` MainMemory MBR register's operation code.

  `6, MMBR_OP, ADD` (At time 6, the MainMemory MBR register 's operation code contains ADD_INST).

- `MMBR_O1:` MainMemory MBR register's operand 1.

  `6, MMBR_O1, 3` (At time 6, the MainMemory MBR register 's operand 1 contains 3).

- `MMBR_O2:` MainMemory MBR register's operand 2.

  `6, MMBR_O2, 5` (At time 6, the MainMemory MBR register 's operand 2 contains 5).

- `XPID:` Contents of the OperatingSystem's variable executingProcessID.

  `34, XPID, 0` (At time 34, the executingProcessID contains 0).

- `RMEM`: Contents of the specified relative memory position of the executing process:
  `5, RMEM, 13, 3` (At time 5, the contents of relative address 3 is 13).

- `AMEM`: Contents of the specified absolute memory position:
  `5, AMEM, 127, 140` (At time 5, the contents of absolute address 140 is 127).

- `MBR:` Contents of the MBR register of MainMemory.

  `6, MBR, 46` (At time 6, the MainMemory MBR register contains 46).

- `MMBR:` Contents of the MBR register of MainMemory.

  `6, MMBR, 53` (At time 6, the MainMemory MBR register contains 53).

- `PCB_ST:` Contents of the status field of the PCB whose index is specified as address.

  `15, PCB_ST, 0, 1` (At time 15, the status field of PCB with PID 1 contains 0).

- `PCB_PC:` Contents of the PC field of the PCB whose index is specified as address.

  `6, PCB_PC, 16, 0` (At time 6, the PC field of PCB with PID 0 contains 16).

- `PCB_PR:` Contents of the priority field of the PCB whose index is specified as address.

  `6, MMBR, 100, 2` (At time 6, the priority field of PCB with PID 2 contains 100).

Assertions file generation

The assertions file can be built manually, as a normal text file, entering an assertion (like the ones above) in a different line.

The assertions file can also be generated automatically, invoking the simulator with option section `--generateAsserts` in the command line. An assertions file is also necessary in this case, but the *`value`* portion of each assertion will be ignored, and a similar assertion will be generated in the standard output with the actual *`value`* for the checked element. For example, in an assertion like the following is present in the assertions file:

`6, PC, `*`any_value`*

The simulator will generate a similar assertion:

`6, PC, `*`value`*

but being *`value`* the actual value of the program counter at time 6. *`any_value`* in the original assertion is not really used, but must be the same type as *`value`*.

Generated assertions are shown on the screen, so the user must copy-and-paste the desired ones in the final *asserts* file.

Time specification

When generating assertions automatically, the “*” character can be used for the `time` specification: in this case, an assertion like the considered will be generated for every time unit.

When using the “*” as a `time` specification when checking assertions, you mean a given element should not change its value throughout the simulation.