Universidade de Aveiro

Mestrado Integrado em Engenharia de Computadores e Telemática

Arquitectura de Computadores Avançada

Lesson 3: Hazard detection

Academic year 2015/2016 — Nuno Lau/José Luís Azevedo

SystemC is a class library and a methodology, based on C++, that allows the description of systems with concurrency, reactivity and latency in the execution of operations. Hence, using SystemC, it is possible to model and simulate systems in their hardware and software components. A description of the basic concepts of SystemC necessary to carry out this assignment is presented below.

Each component of the system to be modelled is represented by a **module**, which can contain input, output or bidirectional **ports**. The interconnection between the ports of a module is carried out by means of **signals**. The behavior of a module is defined through the use of **methods** or by using other modules previously defined. Each method is invoked whenever any of the signals of its **sensitivity list** changes its value. Below is the description, in SystemC, of a 2-input and a 3-input AND gate:

```
SC_MODULE(and) {
  sc_in < bool > a, b;
  sc_out< bool > out;

  SC_CTOR(and) {
      SC_METHOD(entry);
      sensitive << a << b;
  }
  void entry();
};

void and::entry()
{
  out.write(a.read() && b.read());
}
```

```
SC_MODULE(and3) {
  sc_in < bool > a, b, c;
  sc_out< bool > out;

  sc_signal <bool> aux;

  and *a1, *a2;

  SC_CTOR(and3) {
     a1 = new and("a1");
     a1->a(a); a1->b(b);
     a1->out(aux);
     a2 = new and("a2");
     a2->a(aux); a2->b(c);
     a2->out(out);
  }
};
```

or, in a schematic way:

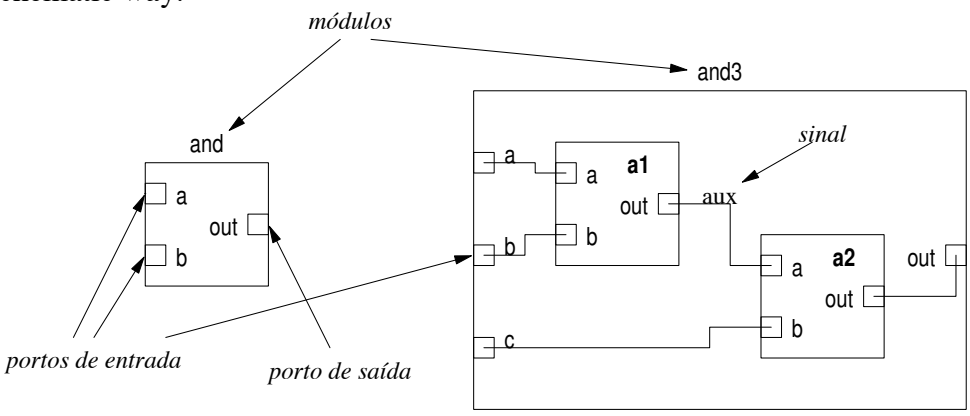

Figure 1

The program **MIPS_SystemC** uses SystemC to simulate an architecture similar to that shown in Figure 6.30 of the book Computer Organization & Design, H & P. The source code of this program is in the archive file **MIPS_SystemC_v0.6.6.tgz** available on the course website, along with the compiled version of SystemC libraries (32bits and 64bits).

The simulated architecture is shown in Figure 2 (see **mips.h** and **mips.cpp** files). The names of the modules (and signals) shown in the figure are identical to those used in SystemC code. This architecture allows the execution of a subset of the MIPS processor instructions.

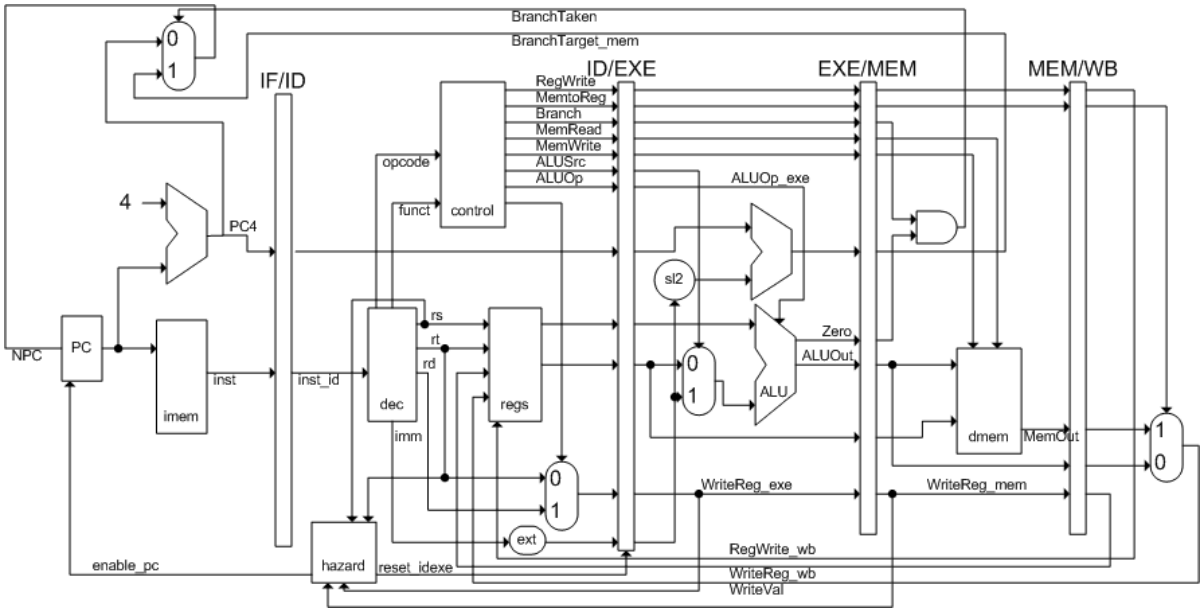

Figure 2

The simulator has a graphical interface that allows viewing the values of the signals inside the processor, as well as the values of the processor registers, data memory and instruction memory, during the execution of a program.

The initial contents of instruction and data memories can be defined by reading files similar to **instmem.hex** and **datamem.dat**, respectively. File **instmem.hex** contains, in each line, the 32-bits hexadecimal machine code of an instruction. The **datamem.dat** file contains 32-bits hexadecimal integer values to be loaded in data memory, sequentially, starting at address **0x00000000**. To change the assembly program to simulate or the contents of data memory, you must edit the files **instmem.hex** and **datamem.dat**, respectively, or create new files, with the same format, and load them into **MIPS_SystemC**.

The file **refman.pdf** is the reference manual for the **MIPS_SystemC** simulator that thoroughly describes the functionality of each of the components of the simulated model.

1. The following machine code, when executed in the provided **MIPS_SystemC** simulator, produces wrong results. Identify the error and its source and correct the architecture. Tip: the error lies in the **mips.cpp** file.

instmem1.hex file:

```
0x8c010000   # lw  r1,0(r0)
0x8c020004   # lw  r2,4(r0)
0x00222020   # add r4,r1,r2
0x00222822   # sub r5,r1,r2
0x0022302a   # slt r6,r1,r2
0x00223824   # and r7,r1,r2
```

datamem1.dat file:

```
0x10       # [0x00]
0x5        # [0x04]
```

2. Determine the number of bits of each of the pipeline registers in the **MIPS_SystemC** simulator.

3. Run the following machine code in the **MIPS_SystemC** simulator. Justify each of the stalls introduced by the detection of hazards. Explain how the number of stalls could be reduced (without using the forwarding technique) while keeping a generic and correct execution. Perform any necessary changes to the simulator so that the code is executed with fewer stalls.

```
instmem3.hex file:
0x8c010000   # lw  r1,0(r0)
0x011020     # add r2,r0,r1
0x8c621000   # lw  r2,0x1000(r3)
```

4. The hazard detection unit (**hazard.h**, **hazard.cpp** and **mips.cpp** files) of the provided **MIPS_SystemC** simulator only detects data hazards. Modify it so that control hazards can also be detected, allowing programs that include the instruction **beq** to run correctly. Use, as test code, the following program:

```
instmem4.hex file:
0x00002020 #       add r4,r0,r0
0x8c010000 #       lw  r1,0(r0)
0x8c050004 #       lw  r5,4(r0)
0x8c030008 #       lw  r3,8(r0)
0x8c07000c #       lw  r7,c(r0)
0x8c620000 # lab1: lw  r2,0(r3)
0x00442020 #       add r4,r4,r2
0x00611820 #       add r3,r3,r1
0x0065302a #       slt r6,r3,r5
0x10c7fffb #       beq r6,r7,lab1
0xac640000 #       sw  r4,0(r3)
0x00002020 #       add r4,r0,r0
0x00001820 #       add r3,r0,r0
```

```
datamem4.dat file:
0x4      # [0x00] integer increment
0x38     # [0x04] after last el addr
0x10     # [0x08] first element addr
0x1      # [0x0C] const used in branch
0x001000 # [0x10] begin array
0x001001 # [0x14]
0x001002 # [0x18]
0x001003 # [0x1C]
0x001004 # [0x20]
0x001005 # [0x24]
0x001006 # [0x28]
0x001007 # [0x2C]
0x001008 # [0x30]
0x001009 # [0x34] end array
0x000000 # [0x38] res. stored here
```

Appendix

This appendix presents the meaning of some common errors that may occur when using SystemC, namely, compilation, execution and model specification.

Compilation errors

Using templates incorrectly: the use of templates is common in the construction of a SystemC model; however, there should be some caution in the way they are declared:

```
mux< sc_uint<32> > *mPC; //correct
mux< sc_uint<32>>  *mPC; //wrong (should have a space between >>)
```

Unsing undefined signals, ports or modules: all signals, ports or modules must be defined before being used. For example, an attempt to use port **`diin`** (not defined) of the **`alu`** module can lead to the following error message:

```
mips.cpp:138: no matching function for call to
              alu::diin(sc_signal<sc_dt::sc_uint<32> >&)
```

Type mismatch between a signal and a port: the connection between a signal and a port is only possible if they are of compatible types. Failure to comply with this rule leads to error messages like this:

```
mips.cpp:129: no match for call to `(sc_out<sc_dt::sc_uint<32> >)
             (sc_signal<sc_dt::sc_uint<3> >&)
```

Execution errors

During the execution of the model, SystemC runs some checks looking for model consistency. When an error is detected, SystemC writes an error message and terminates the program. The most common error situations are listed below.

No signal connected to a given port: all ports of a module must be connected to a signal. The following message indicates that the third port (in order of declaration) of the **`MIPS.alu`** module is not connected to any signal:

```
Error: (E109) complete binding failed: port not bound:
              port 'MIPS.alu.port_3' (sc_out)
```

More than one signal connected to the same port: only one signal can be connected to a given port; attempts to connect more than one signal at the same port lead to messages like:

```
Error: (E107) bind interface to port failed: maximum reached:
              port 'MIPS.alu.port_3' (sc_out)
```

Specification errors

Sensitivity list incomplete: when changing the functionality of a module described in a behavioral way (**`SC_METHOD`**), especially when new input signals are added, the sensitivity list should always be checked and/or updated.