```
1 $ make
2 $ make run
```

Fig. 5.8 Executing the platform

initialized by calling its `init` method (Line 19). The simulation is launched as usual in Line 22. After the platform simulation is concluded, the PowerPC's `PrintStat` method is called (Line 24) to show statistics on the screen, before the simulation process is terminated (Line 25).

To run the executable representation of the platform, two commands are needed: one to build the representation, another to execute it, as shown in Fig. 5.8. The resulting screen, exhibited at simulation completion, is shown in Fig. 5.9. Note that the welcome messaged is printed 10 times, as expected from Fig. 5.6.

Although some refinements could be done on this platform, it is kept as is for simplicity. We leave such improvements to the next section, where a dual-core platform will be built through iterative refinements.

5.3.2 Dual Core Platform

A pretty simple platform, derived from the previous one, is described in Fig. 5.10. It consists of a PowerPC (Line 3), a TLM memory (Line 4), and a simple router

```
1                SystemC 2.2.0
2          Copyright (c) 1996-2006 by all Contributors
3                       ALL RIGHTS RESERVED
4  ArchC: Reading ELF application file: hello_ppc.x
5
6  ArchC: ------- Starting Simulation ----------
7
8  Hi from processor PowerPC!
9  Hi from processor PowerPC!
10 Hi from processor PowerPC!
11 Hi from processor PowerPC!
12 Hi from processor PowerPC!
13 Hi from processor PowerPC!
14 Hi from processor PowerPC!
15 Hi from processor PowerPC!
16 Hi from processor PowerPC!
17 Hi from processor PowerPC!
18 ArchC: ------- Simulation Finished ----------
19 SystemC: simulation stopped by user.
20 ArchC: Simulation statistics
21     Times: 0.01 user, 0.03 system, 0.00 real
22     Number of instructions executed: 17571
23     Simulation speed: (too fast to be precise)
```

Fig. 5.9 Screen capture—Hello World

```
1  int sc_main(int ac, char *av[])
2  {
3    powerpc ppc_proc1("ppc1");
4    ac_tlm_mem mem("mem", 8 * 1024 * 1024);
5    ac_tlm_router router("router");
6
7    ppc_proc1.MEM_port(router.target_export);
8    router.MEM_port(mem.target_export);
9
10   ppc_proc1.init(ac, av);
11   sc_start(-1);
12
13   ppc_proc1.PrintStat();
14   cerr << endl;
15
16   return ppc_proc1.ac_exit_status;
17 }
```

Fig. 5.10 ArchC 2.0 TLM simple single core processor-router-memory platform

```
1  class ac_tlm_router :
2    public sc_module,
3    public ac_tlm_transport_if
4  {
5  public:
6    ac_tlm_port MEM_port;
7    sc_export< ac_tlm_transport_if > target_export;
8
9    ac_tlm_rsp transport(const ac_tlm_req &request)
10   {
11     ac_tlm_rsp response;
12     response = MEM_port->transport(request);
13     return response;
14   }
15   ac_tlm_router(sc_module_name module_name);
16   ~ac_tlm_router() {};
17 };
```

Fig. 5.11 TLM router, first version

(Line 5). Note that the main difference between this platform and the previous one is the presence of a router. The platform's components are connected through TLM binding at Lines 7–8. The router was built such that all TLM requests are simply passed from the processor to the memory and all memory responses are merely returned to the processor. As a result of full router transparency, this platform can run exactly the same code as the previous one.

As shown in Fig. 5.11, the router is a very simple module, which essentially consists of two TLM ports. Note that the router is very similar to the memory described in Fig. 5.3, except from the fact that it passes all incoming requests to the output port without processing (as opposed to the memory functionality), as can be seen in the

```
1  int sc_main(int ac, char *av[])
2  {
3    sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",
         SC_DO_NOTHING);
4    //!  ISA simulator
5    powerpc ppc_proc1("ppc1"), ppc_proc2("ppc2");
6    ac_tlm_mem mem("mem", 8 * 1024 * 1024);
7    ac_tlm_router router("router");
8
9    char *av1[] = {"dual_ppc.x", "--load=hello_dual.x", ""};
10   int ac1 = 3;
11   char *av2[] = {"dual_ppc.x", "--load=hello_dual.x", ""};
12   int ac2 = 3;
13
14   ppc_proc1.MEM_port(router.target_export1);
15   ppc_proc2.MEM_port(router.target_export2);
16   router.MEM_port(mem.target_export);
17
18   ppc_proc1.init(ac1, av1);
19   ppc_proc2.init(ac2, av2);
20   ppc_proc1.set_instr_batch_size(1);
21   ppc_proc2.set_instr_batch_size(1);
22
23   sc_start(-1);
24
25   ppc_proc1.PrintStat();
26   ppc_proc2.PrintStat();
27   return ppc_proc1.ac_exit_status + ppc_proc2.ac_exit_status;
28 }
```

Fig. 5.12 Dual core platform, with two PowerPC processors

`transport` method (Line 9), more specifically at Line 12. To run this platform, it is necessary to add its description to ARP's `is` directory and to include the line `IS := ac_tlm_router` in the file `defs.arp` (described in Fig. 5.5).

Now it is time to add a second processor to the derived platform so as to build a dual-core system. The first modification must be done at the platform `main.cpp` file, as shown in Fig. 5.12. First, the two processors (`ppc1` and `ppc2`) are declared at Line 5. Then, we included two command line parameters, one for each of the processors (Lines 9–11). For the time being, the parameters are exactly the same: they are serving as placeholders to put two programs to run, one for each processor (to actually run two different programs, they must be placed in distinct memory addresses). Then, the previous version of the router was extended (Lines 14–15) with an extra port (`target_export1` and `target_export2`). After processor model initialization (Lines 18–19) and immediately before simulation launching, a same parameter was changed in both processor models (Lines 20–21): `set_instr_batch_size` was set to 1 so as to allow each underlying ISS to alternately run one instruction at time, before calling a `wait` to switch to the other ISS (usually, each ArchC ISS runs 200 instructions before switching; by setting that parameter to 1, the simulation more closely resembles the actual parallel execution

```
1 //!Generic begin behavior method.
2 void ac_behavior(begin)
3 {
4   /* Here the stack is started */
5   GPR.write(1, AC_RAM_END - 1024);
6   ...
7 }
```

Fig. 5.13 PowerPC original `ac_behavior(begin)` method

at the expense of slowing down the simulation; feel free to play with this parameter for a tradeoff between speed and precision).

After running this platform, the welcome message *"Hi from processor PowerPC!"* appears 20 times on the screen, as expected.

At this point, there are two pending issues to be solved for this platform: the two processors have exactly the same stack and are running exactly the same code. Let us address each of them at a time.

The initial value for the stack pointer can be set in two different places: in the startup assembly code or in the processor model description. If the stack pointer value is set in both places, the startup assembly code will prevail, as expected in a real platform. To make platform initialization flexible enough for further potential improvements, we prefer to make changes in the processor model. In the processor description, the place to set the initial value for the stack pointer is in the `ac_behavior(begin)` method, as illustrated in Fig. 5.13, where register 1 (the stack pointer) is written with the address `AC_RAM_END - 1024`. The 1024-byte offset represents reserved memory space through which the simulator can pass command-line parameters to the platform.

Let us now show how we can provide multiple stacks by modifying the code in Fig. 5.13 by defining a stack size and allocating successive stack blocks for each processor. Fig. 5.14 illustrates the required changes. First, the default stack size is defined (256 KB) and the number of active processors is initialized. When a processor model is activated, its the stack pointer is assigned the address of a 256 KB block depending on the number of processors activated so far. In this way, we can automatically handle an arbitrary amount of processors up to the declared memory limit. Since all the processors share the same memory, they can pass pointers from the stack to each other.

Now it is time to make the processors run different code. Although this could be done by loading different codes to different address spaces, we are going to solve this problem in a straightforward way, by loading exactly the same code but splitting the processors execution flow on the fly. Like when solving the previous issue, this is a generic solution that can be applied to an arbitrary number of processors.

We start by creating a new component (`ac_tlm_lock`), which acts as a hardware lock. We reuse exactly the same code as `ac_tlm_mem` by changing only the `readm` and `writem` methods. Figure 5.15 shows the modified method (it is even simpler than the memory's). The `writem` method only assign the `value` private variable and the `readm` returns its value and change it to 1. By so doing, we can

```
1  #define DEFAULT_STACK_SIZE (256 * 1024)
2  static int processors_started = 0;
3  ...
4  //!Generic begin behavior method.
5  void ac_behavior( begin )
6  {
7    /* Here the stack is started */
8    GPR.write(1, AC_RAM_END - 1024 - processors_started++ *
         DEFAULT_STACK_SIZE);
9  }
```

Fig. 5.14 PowerPC original `ac_behavior(begin)` method

```
1  ac_tlm_rsp_status ac_tlm_lock::writem( const uint32_t &a ,
       const uint32_t &d )
2  {
3    value = d;
4    return SUCCESS;
5  }
6
7  ac_tlm_rsp_status ac_tlm_lock::readm( const uint32_t &a ,
       uint32_t &d )
8  {
9    d = value;
10   value = 1;
11   return SUCCESS;
12 }
```

Fig. 5.15 Lock module `readm` and `writem` methods

use this component as a simple hardware lock, which can be checked by the same software code to split the program flow.

After creating that component, the next step is to connect it to the platform. This can be easily performed inside our router, by creating a new `ac_tlm_port` called `LOCK_port` (after Line 6 of Fig. 5.11) and changing the `transport` method as in Fig. 5.16. To distinguish between ports, an `if` statement is included in Line 5. Selection is based on the request address. Note that the address 0x800000 is adopted for the lock component. When a request is issued with that address, the lock component receives the packet. Note that, every time a new component is needed, it is only a matter of including a new (nested) `if` statement for selection purposes and adding a new output port in the router. This make this router easily scalable.

To show that both pending issues were solved by the described mechanisms, we are now going to change the "Hello World" program. Figure 5.17 shows the changes. The first change is the inclusion of a `RecursiveHello` (Line 3) function, which recursively calls itself several times. We do so to show that the processors actually have different stacks. To use the lock hardware, we add a pointer to the 0x800000 address. The `AcquireLock` just has to wait until this pointer has a zero value (remember that for every hardware read, the value is changed to one). The `Release-`

```
1  ac_tlm_rsp transport( const ac_tlm_req &request ) {
2
3    ac_tlm_rsp response;
4
5    if (request.addr != 0x800000)
6        response = MEM_port->transport(request);
7    else
8        response = LOCK_port->transport(request);
9
10   return response;
11 }
```

Fig. 5.16 New transport method for the router

```
1  #define STARTUP_ADDRESS 0x800000;
2  volatile int procCounter = 0;
3  void RecursiveHello(int n, int procNumber)
4  {
5      if (n) {
6          printf("Hi from processor PowerPC %d!\n",
               procNumber);
7          RecursiveHello(n - 1, procNumber);
8      }
9  }
10
11 int main(int ac, char *av[]){
12   int procNumber;
13
14   AcquireLock();
15   procNumber = procCounter;
16   procCounter ++;
17   ReleaseLock();
18
19   if (procNumber % 2) {
20       for (i = 0; i < 100000; i ++);
21   }
22   RecursiveHello(10, procNumber);
23   exit(0);
24   return 0;
25 }
```

Fig. 5.17 New Hello World program for a dual core example

`Lock` is as simple as assigning a zero to the lock pointer. These two functions are shown in Fig. 5.18. At Fig. 5.17, Line 15, each processor gets a different number by reading the incrementing the global `procCounter` variable. Notice that this line is protected by the lock. In Lines 19–21, we had to implement a delay loop because the `printf` function we used is not reentrant[1] (if we remove the delay loop, the

[1] A reentrant function is a function that can be called more than once at the same time, providing the expected behavior. Usually, this is accomplishing by not using any global buffer.

```
1  #define LOCK_ADDRESS 0x800000;
2  volatile int *lock = (volatile int *) LOCK_ADDRESS;
3
4  void AcquireLock()
5  {
6      while (*lock);
7  }
8
9  void ReleaseLock()
10 {
11     *lock = 0;
12 }
```

Fig. 5.18 `AcquireLock` and `ReleaseLock` implementations

welcome message will be printed 20 times, but the messages from each processor will be mixed in the output screen).

The screen resulting from running the executable platform representation is shown in Fig. 5.19. Notice that, due to the delay loop, one processor first prints the welcome message 10 times and, only after that, the other processor prints yet another 10 times.

Although this solves the reentrance problem, it is not what we expected by running two processors at the same time. Let us change the code to have the opportunity of interleaving the `printf` functions. To do so, we need to have some concurrence control: we added a new function (`PrintHello`) to print the welcome message and changed the Line 6 of Fig. 5.17 to call it. The `PrintHello` implementation is shown in Fig. 5.20. Notice that we have just called `AcquireLock()` before and `ReleaseLock()` after the `printf`. By running this new code, the 20 welcome messages (10 from processor 0 and 10 from processor 1) will result interleaved.

5.4 The MP3 Example

Up to now, we have seen just a "Hello World" platform. Now we will put together one full example, an MP3 decoder based on a dual core platform. Although decoding a MP3 is not a processor-hungry task, it is a very good example of software that can be partitioned between hardware and software to serve as an example. We will use the MP3 implementation called `dist10`[2].

5.4.1 Profiling

The first required step is to profile the MP3 software decoder to do a hardware-software partitioning. We will use the `gprof`. To use profiling with gprof, the de-

[2]This implementation is considered a base implementation with focus on readability instead of speed.