2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Automate Interrupt Checking with UVM Macros and Python

Guide With Code Examples

Aleksandra Dimanic, Vtool Ltd, Belgrade, Serbia (aleksandrad@thevtool.com)

Nemanja Stevanovic, Vtool Ltd, Belgrade, Serbia (nemanjas@thevtool.com)

Yoav Furman, Chain Reaction Ltd., Yokneam, Israel (yoavf@chain-reaction.io)

Itay Henigsberg, Chain Reaction Ltd., Yokneam, Israel (itayh@chain-reaction.io)

***Abstract*—This paper will present a macro-based system for verifying interrupts. The main goal is to create a reusable and automated system which can later be generated using Python scripts.**

Keywords—verification, interrupts, reusable, python

I. Introduction

Interrupts are an essential part of every design, and must be thoroughly tested to make sure that the device can recover from unexpected failures, or that it can properly handle asynchronous events which are in some way important for the system. The main goal of this paper is to help verification engineers create a more general and reusable way of checking the interrupts, and to automate parts of the process using Python scripting in order to save time and effort. This approach can help companies standardize and set up methodology for verifying the functionality of interrupts.

The method described in this paper has been used in an ongoing project with Chain Reaction Ltd, alongside Vtool Ltd (design and verification service company). The person reading this paper should be familiar with the verification methodologies and SystemVerilog in order to understand the provided solutions and code examples.

This paper will cover handling and checking the interrupt trigger via a dedicated interface, as well as status and control registers using macros. The focus is on an automatic way of generating code for checking interrupts using Python scripts, according to the main building blocks of code.

II. Interrupt overview

All systems share the same common ground for IRQ (interrupt) handling. An interrupt is triggered by the design, according to the system specification. When the condition for the interrupt is satisfied, it is reflected in the appropriate status registers. Aggregating the IRQ to software or interrupt handler takes into account mask/unmask/set/reset registers. After getting an IRQ request, the system needs to take specific steps in order to get back to normal mode of operation. This is unique for each interrupt and is out of the scope of this paper.

Interrupts are usually grouped in blocks within the design, meaning they share common paths in RTL (register-transfer level). They also share common behavior - when there is an interrupt trigger, one should send an interrupt request. The idea is to take those common parts, and create a general checker in order to save time and effort.

III. Explaining the concept

Each interrupt has its corresponding register or a field in the register. A group of interrupts is mapped in the ISR (Interrupt Status Register) register, where each field correlates to a specific interrupt handler. Reading from

status registers can be done at any point while the system is active, meaning that the registers always have to reflect the correct values of the IRQs.

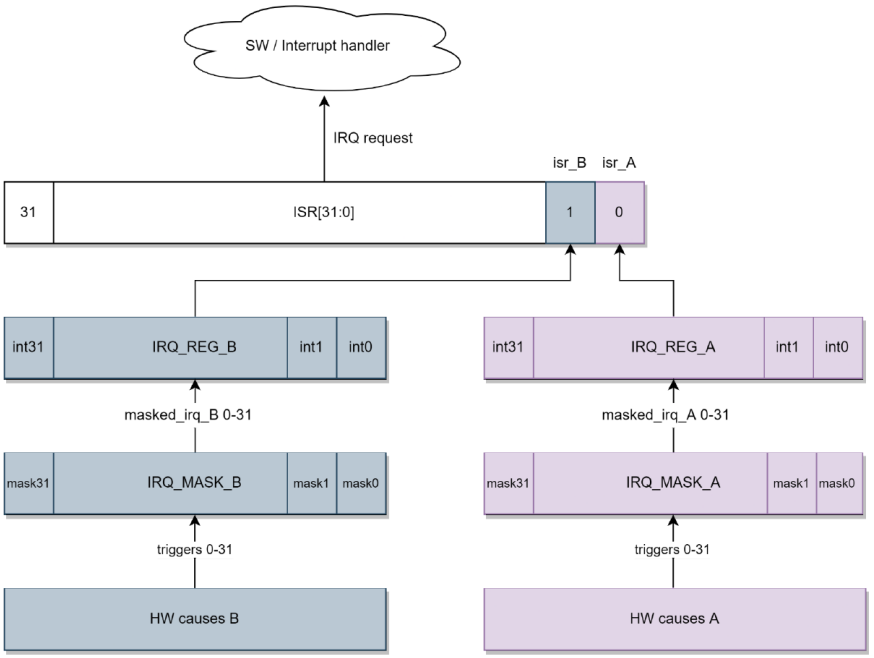

Figure 1: Interrupt registers example

More often than not, verification is not completely in sync with the RTL, meaning that sometimes we know what behavior to expect from the RTL before it actually happens, and sometimes we know what to do only after it happens. In order to avoid mismatches in RTL and verification, and to avoid dependencies between checkers and register prediction, we can create a general task which will always do the register prediction for us, based on the interrupt trigger coming from the design. The general predict mechanism can be used to check the aggregation of the interrupt, and can be fully automatic, but in order to completely verify the proper functionality of IRQs, it is also necessary to check the trigger coming from the design.

We need to make sure that the RTL functions properly, meaning checking must be done in both directions. The verification's expectations and RTL behavior must match, i.e. the interrupt trigger has to occur and also be expected. If the trigger was expected in verification and didn't happen in RTL, or if it happened in RTL and it wasn't expected in verification, then that behavior is not correct and either there is a bug, or the verification needs to be fixed. This is where the previously mentioned interrupt trigger checking interface comes into play. The aforementioned interface is configurable, and can detect interrupt triggers, and compare them with verification expectations coming from the reference model (which are out of scope of this paper, as each interrupt is specific). The interface is applicable to all interrupt triggers, both edge and level triggered.

IV. Explaining the flow of interrupt verification

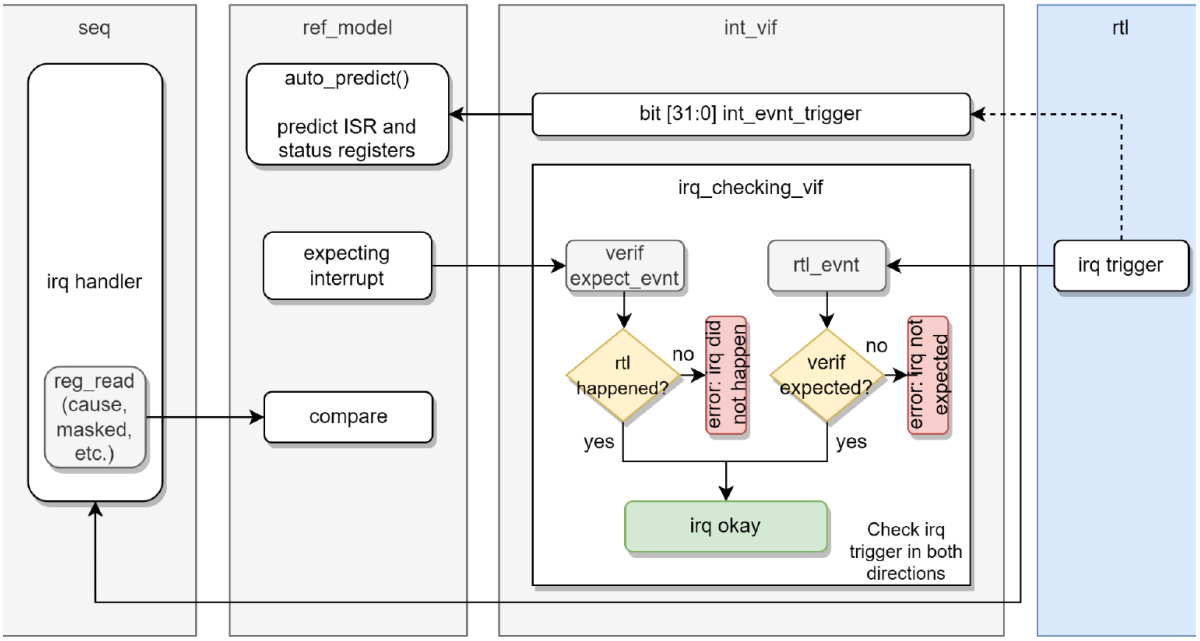

Figure 2: Interrupt verification flow

In order to make the code more uniform first we need to create a new interface to which all relevant IRQ signals will be connected. The interface is called *int_vif* in this paper. All the IRQ triggers coming from the design are connected to a signal in *int_vif* called *int_event_trigger*. The triggers must be connected in the same order as they are mapped in the status registers.

In the reference model, called *ref_model*, there is an *auto_predict()* task. In this task automatic prediction of status registers is performed based on the *irq_evnt_trigger* signals from *int_vif*.

Inside the *int_vif* there is another interface, which is used to check the IRQ triggers, called *irq_checking_vif*. One instance of this interface will check only one trigger. The RTL trigger is connected to a signal called *rtl_trigger*. Once the trigger is detected, the checking begins. The requirement of this interface is for the verification event to be triggered first, so as soon as *rtl_trigger* happens, the interface will first check if the *verif_evnt* was already expected and if it wasn't, the interface will indicate the error.

From the verification side, there is an *verif_evnt* event which needs to be triggered from the *ref_model*. Triggering of this signal is specific for each interrupt, and is out of the scope of this paper. Once the *verif_evnt* is triggered, a verification timer will start. If the timer expires without the *rtl_trigger* happening, the interface will indicate an error. If the *rtl_trigger* happens before the timer expires, it means that the interrupt trigger is behaving properly, and that it was both expected and occurred. If one of the two errors happens, then that means there is an issue either in the design or in the verification.

A. Handling status and control registers

Considering the flow described in section IV and that the same logic is applied to each interrupt, let's take a look at a macro which will do the automatic prediction in the reference model.

```
`define IRQ_AUTO_PREDICT(isr_reg_name, isr_field, irq_reg_name, irq_field,
reg_mask_name, reg_mask)\
forever begin\
    int position =
    reg_model.IRQ_BLOCK.irq_reg_name.irq_event.get_lsb_pos();\ @(posedge
    int_vif.int_event_trigger[position]);\
```

```
    if(!reg_model.IRQ_BLOCK.reg_mask_name.reg_mask.get_mirrored_value()) begin\
        `uvm_info(get_full_name(), $sformatf("Predicting ISR reg due to %s",
reg_model.IRQ_BLOCK.irq_reg_name.irq_event.get_name()), UVM_HIGH)\
         assert(reg_model.IRQ_BLOCK.isr_reg_name.isr_field.predict(.value(1)))
    ;\ assert(reg_model.IRQ_BLOCK.irq_reg_name.irq_event.predict(.value(1)));\
    end\
end\
```

** Note: *In this example the IRQ is masked if the value of the masking register is one.*

The task will detect every rising edge of the interrupt trigger and predict the register values while taking into consideration the masking register. One can notice that the above code can be applied to each interrupt; we just need to pass the proper arguments to the macro.

This prediction is taking 9 lines of code. If for example, a system has 20 different interrupts, without using the macro those 9 lines just became 180. By using the macro we are optimizing the code by avoiding unnecessary duplication. Looking at the register example from figure 1, the *auto_predict()* task will look like this:

```
task auto_predict();
    fork
        //IRQ_REG_A
        `IRQ_AUTO_PREDICT(ISR, isr_A, IRQ_REG_A, int0, IRQ_MASK_A, mask0)
        . . .
        `IRQ_AUTO_PREDICT(ISR, isr_A, IRQ_REG_A, int31, IRQ_MASK_A, mask31)
        //IRQ_REG_B
        `IRQ_AUTO_PREDICT(ISR, isr_B, IRQ_REG_B, int0, IRQ_MASK_B, mask0)
        . . .
        `IRQ_AUTO_PREDICT(ISR, isr_B, IRQ_REG_B, int31, IRQ_MASK_B, mask31)
    join_none
endtask
```

A certain system can have some additional specific behaviors for the interrupts, and by following the same logic more macros can be added.

By using a general task, we can centralize the checking in one location and create an independent standardized mechanism, which will make the debug and implementation process easier, faster and reusable on the same project, but also on company level.

B. How does the irq_checking_vif work?

As mentioned before, one instance of this interface will check one interrupt trigger. Let's look at the flow of checking in one instance of this interface:

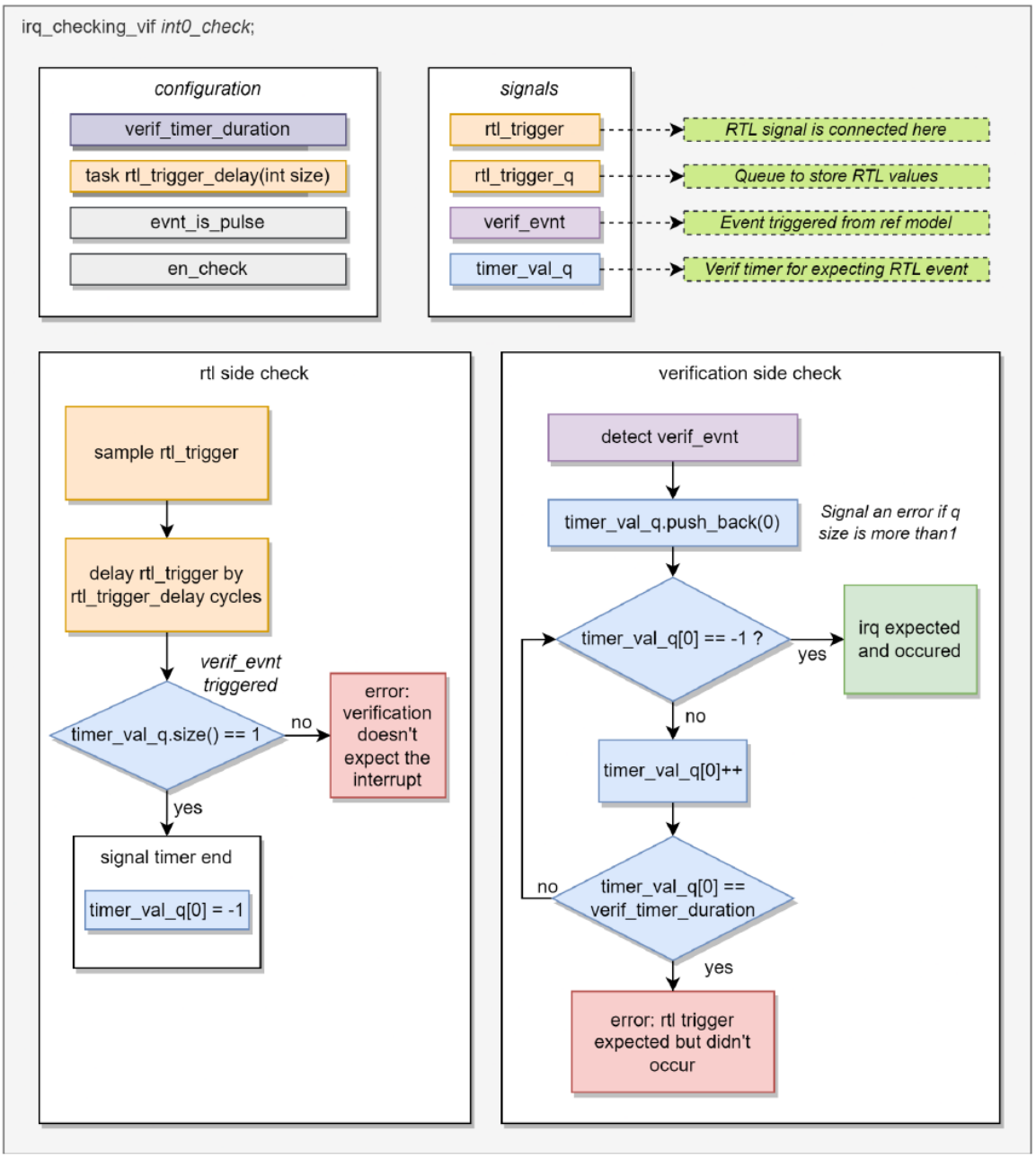

Figure 3: Interrupt checking interface flow diagram

The requirement in this interface is to trigger the verification event before it happens in RTL, and because design and verification are not always in sync, *rtl_trigger* sometimes needs to be delayed. This is why the *rtl_trigger* is continuously sampled and its value is stored in a queue of configurable size called *rtl_trigger_q[$]*. The size of this queue is configured from the reference model, by calling a task *rtl_trigger_delay*.

```
task rtl_trigger_delay(int size);
   if (size == 0) `uvm_error("%m", "Size cannot be zero")
```

```
    repeat(size) rtl_trigger_q.push_back(0);
endtask
```

Sampling and delaying is done in an always block which looks like this:

```
always @(posedge clk) begin
    void'(rtl_trigger_q.pop_front());
    void'(rtl_trigger_q.push_back(rtl_trigger));
end
```

The actual value of the *rtl_trigger* is always stored in the last position in the queue, and everything is being shifted one position to the left. After *rtl_trigger_q[0]* changes from 0 to 1; a new task is called where the occurrence of *verif_evnt* is checked by checking the status of queue *timer_val_q*. If the *verif_evnt* was not triggered, the interface will indicate an error saying that the RTL event happened without it being expected. If the *verif_evnt* was already expected then the value of *timer_val_q[0],* then it will be set to -1.

```
always @(posedge clk) //for pulses __|**|__
    if (evnt_is_pulse && rtl_trigger_q[0])
          check_if_evnt_expected();

always @(posedge rtl_trigger_q[0] iff !evnt_is_pulse ) //for level __|******
          check_if_evnt_expected();

function void check_if_evnt_expected();
    `uvm_info($sformatf("%m"), "rtl event triggered", UVM_HIGH)
    if (en_check)
          if (timer_val_q.size > 0) timer_val_q[0] = -1;
          else `uvm_error($sformatf("%m"), "RTL event occurred without it being expected.")
endfunction
```

Looking at the verification side, after *verif_event* is triggered from the reference model to indicate that an interrupt is expected, a task which monitors the status of *timer_val_q* begins. If *timer_val_q[0]* reaches *verif_timer_duration* without the RTL event happening, the interface indicates an error saying that the event was expected but never occurred. If the RTL event happened, then the *timer_val_q[0]* is already set to -1 and the interrupt is both expected and occurred, meaning that it is functioning correctly.

```
always @(verif_evnt)
     if (en_check && (evnt_is_pulse || rtl_trigger_q[0] === 0)) begin
        if(!timer_val_q.size()) timer_val_q.push_back(0);
        else `uvm_error($sformatf("%m"), "verif event expected more than once")
        `uvm_info($sformatf("%m"), "verif event triggered", UVM_HIGH)
       if (timer_val_q.size == 1)
           fork
              check_expected_ev_happens(); //check if rtl signal happens
          join_none
    end

task check_expected_ev_happens();
    while (timer_val_q.size > 0) begin
```

```
        @(posedge clk);
        if (timer_val_q[0] == -1) begin
            void'(timer_val_q.pop_front());
            `uvm_info($sformatf("%m"), "RTL event expected and occured.", UVM_HIGH)
        end
        if (timer_val_q[0] == verif_timer_duration) begin
            void'(timer_val_q.pop_front());
            `uvm_error($sformatf("%m"), "RTL event didn't occur although expected in
        verification.") end else
           timer_val_q[0] += 1;
    end // while
endtask // check_expect_happens
```

In conclusion, the interface is doing the checking in both directions and making sure that the interrupt trigger is both expected and that it actually happened. There are more additional options that can be added, for example: stopping the checking, ignoring the trigger, checking x/z events on the *rtl_trigger*, etc. The interface allows us to test the interrupts without writing unnecessary code, and to do the checking in a more precise way because the detection window is smaller.

V. Automating everything using Python

Python is a powerful tool which can help engineers speed up the verification process. In this case, a gui based script has been used to automatically generate parts of code based on user input. Let's look at the script output using the example registers shown in figure 4:

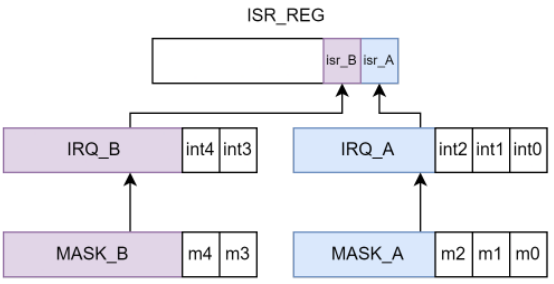

Figure 4: ISR and status register example

After starting the script, the user needs to specify the number of fields in one IRQ status register. For register *IRQ_A* this means inputting number three. A window pops up (figure 5), where the user can input the name of the status register (*IRQ_A*), the names of fields in the status register (*int0, int1, int2*), the name of the masking register (*MASK_A*) and its fields (*m0, m1, m2*), the name of the ISR register (*ISR_REG*) and the name of the field mapped in the ISR register (*isr_A*). By clicking generate, the needed code for register *IRQ_A* is stored in a text file, and after that the user can either close the app or repeat the process for a different register. For the given example the user needs to apply the same logic for *IRQ_B*. The user can also choose if an instance of the *irq_checking_vif* is needed for each interrupt trigger.

After generating the code for all registers, the script can be closed and the entire code can be found in a single text file (figure 6). The user still needs to take the code and move it to proper locations in the verification environment.

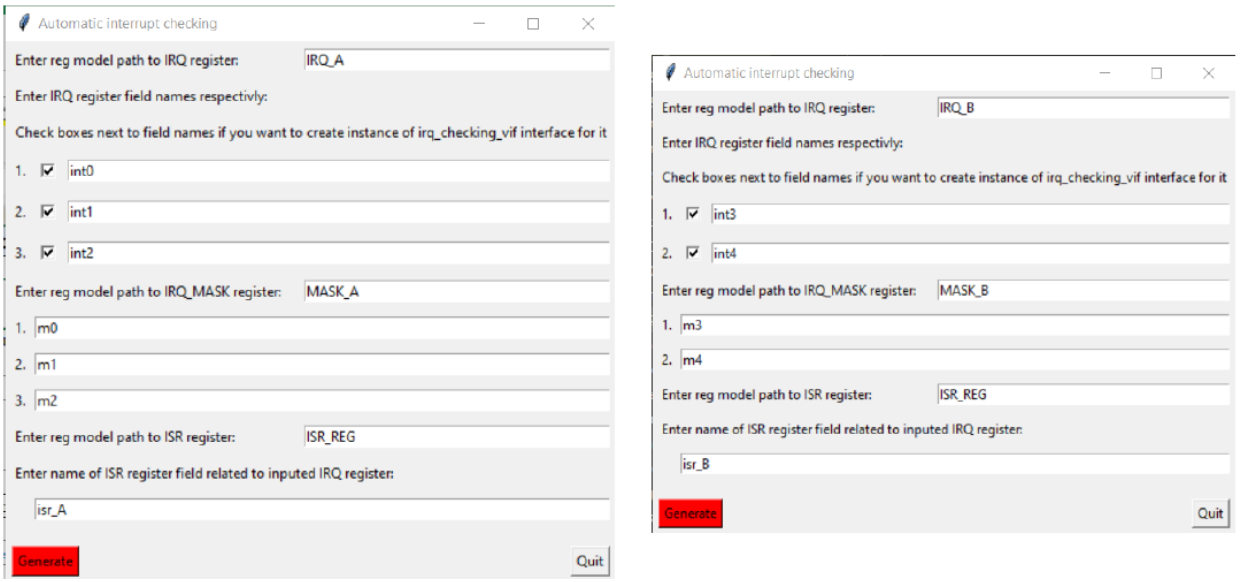

Figure 5: Gui script user interface

```
fork
    `IRQ_AUTO_PREDICT(ISR_REG, isr_A, IRQ_A, int0, MASK_A, m0) //IRQ_A
    `IRQ_AUTO_PREDICT(ISR_REG, isr_A, IRQ_A, int1, MASK_A, m1) //IRQ_A
    `IRQ_AUTO_PREDICT(ISR_REG, isr_A, IRQ_A, int2, MASK_A, m2) //IRQ_A
    `IRQ_AUTO_PREDICT(ISR_REG, isr_B, IRQ_B, int3, MASK_B, m3) //IRQ_B
    `IRQ_AUTO_PREDICT(ISR_REG, isr_B, IRQ_B, int4, MASK_B, m4) //IRQ_B
join_none

irq_checking_vif trigger_check_IRQ_A_int0;
irq_checking_vif trigger_check_IRQ_A_int1;
irq_checking_vif trigger_check_IRQ_A_int2;
irq_checking_vif trigger_check_IRQ_B_int3;
irq_checking_vif trigger_check_IRQ_B_int4;
```

Figure 6: Script output (.txt file)

VI. Results

This method of checking the interrupts has proven effective according to empirical evidence gathered while working on a project with Chain Reaction, with the goal of standardizing the interrupt checking in this and all future projects. The solutions provided in this paper have increased code reusability and decreased the time needed in order to fully verify the interrupts in the SystemVerilog environment. Depending on specific needs of the company or a project, the Python script as well as the macros and the trigger checking interface, can be further expanded or adapted.

VII. Conclusions

Speeding up the verification process is one of the main goals of verification engineers. This paper introduced a standardized method for checking the interrupts and is showing how verification time can be reduced, optimized and partially automated. Presented methodology can increase the quality of any verification environment by improving readability, robustness and reusability. It gives a different perspective, and offers verification engineers one possible solution to overcome some common challenges and problems.