gen_uvm_reg_model

version

Contents

Welcome to gen_uvm_reg_model’s documentation!

Overview

`gen_uvm_reg_model` is a Python based script that can create large scale UVM reg_models, along with software header files and PDF documents.

`gen_uvm_reg_model` builds on various other register flows, like `gen_regs_py` to facilitate a automated flow for design to DV handoff.

Running the Script (Options)

`-h, --help` — Shows the HELP message. Also prints a link to this documenation

-b, -blk (REQUIRED)

Input block/system file to be used for register model generation.

-o, -out (Optional)

Output reg_model name. Defaults to *wav_uvm_reg_model*. Define without any file extension.

-p, -prefix (Optional)

Global prefix to apply to each register. This is usually a chip/project name. Use caution when dealing with lower level blocks.

-aw, -addr_width (Optional)

Sets the **ADDRESS** width for internal reg mappings. Defaults to 32 bit.

-ch, -cheader (Optional)

Prints out a C #defines to a .h file for each block in the system. The output of this is usually passed to the software team.

-dbg, -debug (Optional, not recommended)

Prints out various debug messages during the run. Only use when debugging.

-sp, -script_path (Optional, not recommended)

Changes the search path for `gen_regs_py` or any other scripts called. This is usually for the developer or for trying out a change before submitting it live.

Methodology / Terms

Generally for any SoC, there are a variety of IPs that are created and instantiated. Most, if not all of these IPs will have some level of registers which will be used to communicate or control the IP. Take for example a SPI IP with several registers:

While DVing this block at the IP level may seem trivial, once this IP is part of a larger subsystem the complexity can grow. There may be multiple instances of this SPI IP. There may be other IPs with the same register/bitfield names. Testing and managing these registers can begin to be quite complicated.

`gen_uvm_reg_model` aims to help simplify this by allowing IP and subsystems to define BLK/SYS (block/system) files which they will use for DV at thier IP/subsystem level testing, and can be pulled in at the next level of hierarchy, even up to chip top. This allows an IP developer to clearly define register instances and addressing for the next level of integration.

`gen_reg_uvm_model` uses the following terms when describing various register components:

```
System1
  --Register Block1
    --Register1
    --Register2
  --Register Block2
    --Register1
    --Register2
```

Registers / Bitfields

We will classify regsiters as a typical 32bit register which software can access. Each register is comprised of bitfields.

Register Block

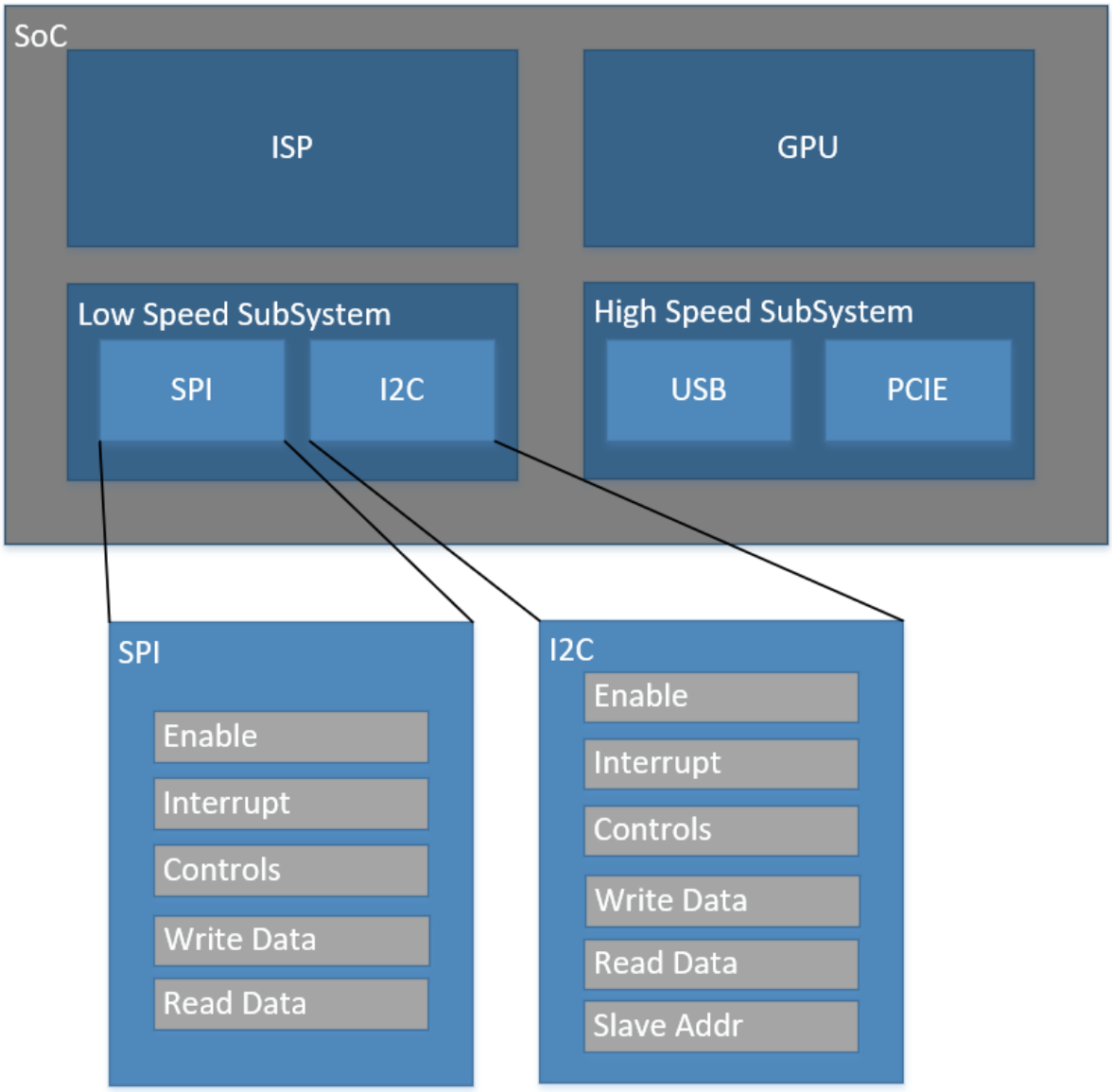
This is the level in which a set of registers has been described. *Usually* this is for registers which are created using `gen_regs_py`, although any collection of regsiters is considered a `Register block`

System / SubSystem / Register Subsystem

This is a level in which one or more `register blocks` have been instantiated to form a `System`.

Top Level

This is the highest level of the register system. It is the DUT.



Taking the above image as an example, we would say the following:

- SPI is a `Register Block` with some registers `Enable`, `Interrupt`, etc.
- I2C is a `Register Block` with some registers `Enable`, `Interrupt`, etc.
- Low Speed Subsystem is a `System` with `register blocks` SPI and I2C.
- High Speed Subsystem is a `System` with `register blocks` USB and PCIE
- SoC is a `System` that includes the other `Systems`, “Low Speed Subsystem” and “High Speed Subsystem”, as well as two other Systems. In this case the SoC is the `Top Level`.

Addressing

`gen_uvm_reg_model` works on the premise that all regsiter blocks have a base address of 0x0000. When a register block is instantiated in a design, a designer would add in decode logic which would then “assign” an address to that group of registers. The block/system files, are a description of this addressing scheme. Because of this, it is generally recommended that designers either create this file or at least signoff on it’s contents.

There will be checks to validate the reg_model vs. the design, which should flag any issues, however the initial setup needs to be completed with an understanding of how the DUT has been architected.

> ***Tip***
>
> When a particular IP has been created with multiple register blocks and is instantiated at a higher level, the IP developer cannot be sure of the base address for their IP. Because of this, it is generally good design practice to create your block/system files with the start address beginning at 0x0000_0000

Input File Types

When a user runs `gen_uvm_reg_model`, they pass the `top level` block/system file to the script. This top level file will describe all registers that are to be included in the DUT.

In general, the user will create the top level file by describing various register blocks, with unique names, and address offsets. A user would "instantiate" a register block by simply providing path to the file that is used for `gen_regs_py`, or other register generation tools.

TL;DR

Too Long, Didn't Read. If you just want a quick "how do I do..." in the block file:

```
MAP:<Name>                                                              <Starting Address>
BLK:<File>        <Instance Name>      <Block Prefix>  <Block Name>       <Address Offset>
SWI:<File>        <Instance Name>                                        <Address Offset>
DV:<File>         <Instance Name>                                        <Address Offset>
SYS:<File>        <System Name>                                          <Address Offset>
```

Block/System File Directives

Environment Variables

Standard Linux Environment Variables can be used inside the block/system files to "neaten" up the files. Normal use cases would be to use an env variable for the starting location of the workdir where the registers can be found. For example, you have your register in the following directory:

```
/projects/somechip/john117/regs
  |--/projects/somechip/john117/regs/spi
    |--/projects/somechip/john117/regs/spi/spi_regs.txt
  |--/projects/somechip/john117/regs/i2c
    |--/projects/somechip/john117/regs/i2c/i2c_regs.txt
```

You may want to set a `SOMECHIP_REGS` environment variable to `/projects/somechip/john117/regs`.

Environment Variables will follow the `${<VAR>}` syntax. If the variable is not set for the user, an error message will appear.

> ***Caution!***
>
> Use caution when setting ENV variables across multiple projects.

BLK (Blocks)

BLK's are essentially the most basic level for `gen_uvm_reg_model`. They are the same input files as the `gen_regs_py` tool flow. This is done to keep a single source input for RTL, DV, and Software.

When defining a BLK, the following Sytax is used for the block/system file:

```
BLK:<File>        <Instance Name>          <Block Prefix>  <Block Name>       <Address Offset>
```

- `BLK:` - A directive to signal to `gen_reg_uvm_model` to treat this as a register block for `gen_regs_py` to handle
- **File** - File used for `gen_regs_py` with register definitions
- **Instance Name** - This is an instance specific name that is given to this block of registers. `N/A` can be used to denote no instance name, however, instance names are generally *recommended*.

- **Block Prefix** - This is *generally* the same block prefix used for RTL generation with `gen_regs_py`.
- **Block Name** - This is *generally* the same block name used for RTL generation with `gen_regs_py`.
- **Address Offset** - This is the address offset for this register block. Use 0x0000 notation (underscores are allowed)

All of these entries are **REQUIRED**.

Here is an example of a block/system file where I am defining three (3) instances. Two (2) SPI instances, SPI0 and SPI1, and an I2C instance, I2C0. Giving each a 256byte spacing:

```
#BLK:File                          Inst Name    Prefix   Block      Offset
BLK:${TEST_REGS}/spi_regs.txt      SPI0         spi      regs       0x0000
BLK:${TEST_REGS}/spi_regs.txt      SPI1         spi      regs       0x0100
BLK:${TEST_REGS}/i2c_regs.txt      I2C0         i2c      regs       0x0200
```

SWI

A register block in the format of the register input files using SWI format. When an `SWI` file is seen, `gen_uvm_reg_model` will parse the `SWI` file and construct a register block similar to that of `BLK` files.

A user can declare a `SWI` file with the following syntax:

```
SWI:<File>      <Instance Name>         <Address Offset>
```

All of these entries are **REQUIRED**.

> ***Note***
>
> The inputs **File**, **Instance Name**, and **Address Offset** follow the same definitions as for the BLK

DV (DV blocks)

DV components are text files describing register blocks that are *generally* 3rd party IP. These are IPs in which the `gen_regs_py` flow was not used.

A user can declare a DV file with the following syntax:

```
DV:<File>      <Instance Name>         <Address Offset>
```

All of these entries are **REQUIRED**.

> ***Note***
>
> The inputs **File**, **Instance Name**, and **Address Offset** follow the same definitions as for the BLK

Since DV component files are only a last resort, we will not go into the details of their file types. Please see previously used blocks as an example.

SYS

A register system that has at least one BLK/DV/SWI/SYS instance. Can instantiate other SYS.

```
SYS:<File>        <System Name>                                        <Address Offset>
```

- **File** - File with the BLK/DV/SWI/SYS declarations

- **System name** - Name of the system. A prefix is added to all registers under this system with the **System Name**
- **Address Offset** - Base address of this system

This is how a user would build a larger SoC level register block. Let's take the same 3 BLK example from the BLK (Blocks) section.

```
#This is in a file called "pss.blk"
#BLK:File                          Inst Name    Prefix   Block      Offset
BLK:${TEST_REGS}/spi_regs.txt      SPI0         spi      regs       0x0000
BLK:${TEST_REGS}/spi_regs.txt      SPI1         spi      regs       0x0100
BLK:${TEST_REGS}/i2c_regs.txt      I2C0         i2c      regs       0x0200
```

Now let's say I need to instantiate this IP 3 times. I could list each of these separately (6 SPIs and 3 I2Cs) or I can instantiate the `pss.blk` file as a SYS, and denote the base address

```
SYS:pss.blk          PSS0          0x0000    # PSS0_SPI0 -> 0x0000, PSS0_SPI1 -> 0x0100, PSS0_I2C0 -> 0x0200
SYS:pss.blk          PSS1          0x1000    # PSS1_SPI0 -> 0x1000, PSS1_SPI1 -> 0x1100, PSS1_I2C0 -> 0x1200
SYS:pss.blk          PSS2          0x2000    # PSS2_SPI0 -> 0x2000, PSS2_SPI1 -> 0x2100, PSS2_I2C0 -> 0x2200
```

This would generate a register system with where PSS0_SPI0 is at address 0x0000, PSS1_SPI0 is at address 0x1000, and so on.

MAP

Used to create an additional uvm_reg_map. **Any** BLK/DV/SWI/SYS instances after this declaration will use this MAP.

A user can declare a MAP with the following syntax:

```
MAP:<Name>                                                    <Starting Address>
```

- **Name** - Name of the map. Will be declared as a uvm_reg_map <Name> in the reg model
- **Starting Address** - Starting address of this map. ALL BLK/SWI/DV files defined after this are addressed with respect to this map

Example:

```
MAP:MAP_AHB                                                   0xC000_0000    #Start MAP_AHB with base 0xC000_0000
BLK:${TEST_REGS}/spi_regs.txt      SPI0         spi      regs       0x0000         #This is now at 0xC000_0000
BLK:${TEST_REGS}/spi_regs.txt      SPI1         spi      regs       0x0100         #This is now at 0xC000_0100
BLK:${TEST_REGS}/i2c_regs.txt      I2C0         i2c      regs       0x0200         #This is now at 0xC000_0200
```

The **DEFAULT** map, which is instantiated if no MAPs are set, is the MAP_APB.

> ***Warning***
>
> Only the top level block/system file will have maps applied. This means that if you have MAPs defined in lower level SYS files, they will be ignored. Care should be taken when constructing SYS where multiple MAPs are needed at the `Top Level`