Steps to setup RISC-V based SoC verification environment

Agenda

- Introduction
- Creating an SoC verification environment
  - Block diagram
  - Three steps to setup the environment
- UVM architecture
- Converting C programs to binary files
  - Compiling C program : behind the scene
  - Pre-processing
  - Compilation
  - Assembly and linking
    - start.S file
  - Loader
  - How to setup interrupts
- Synchronizing C programs and UVM tests
- Summary

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Introduction

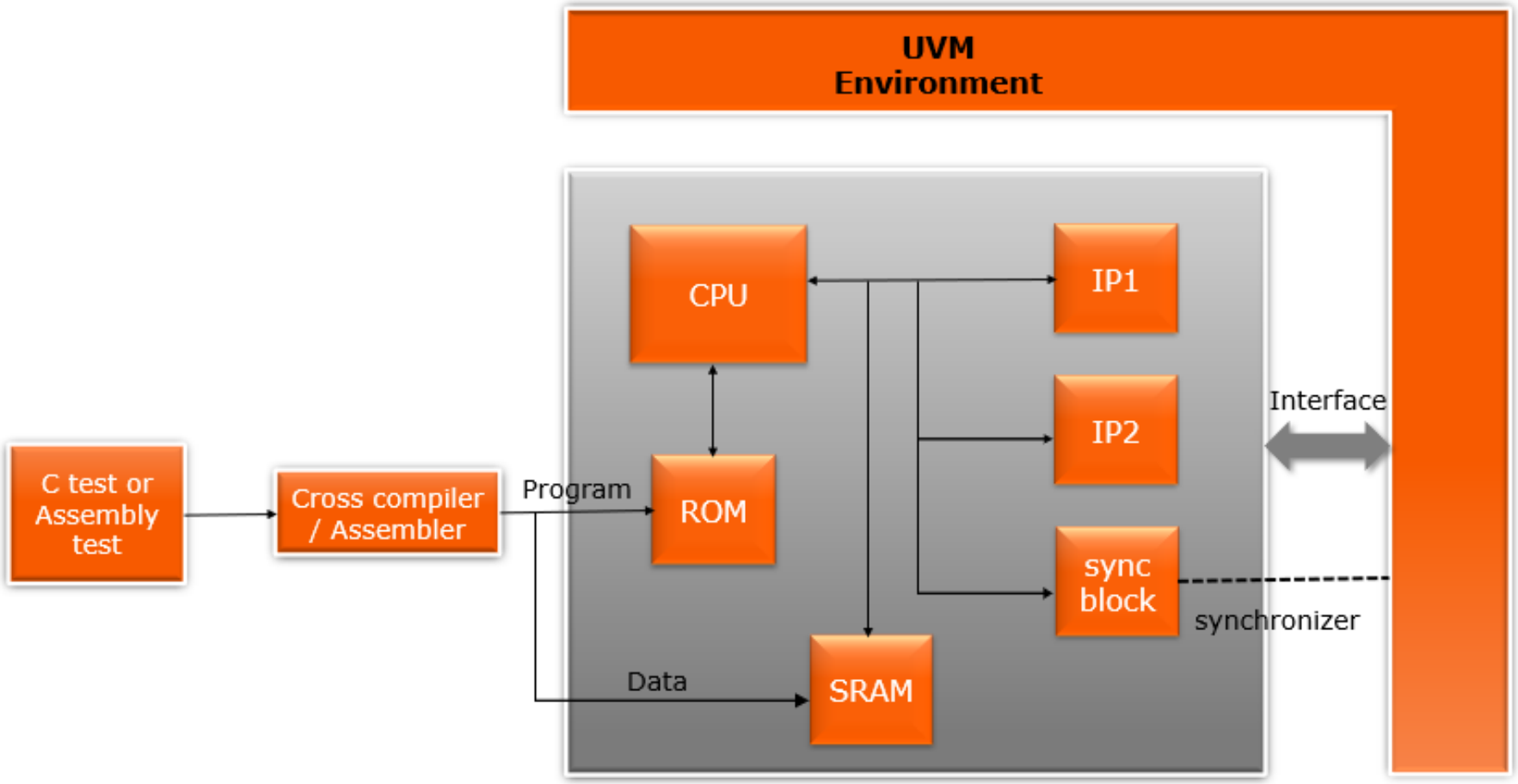
Introduction : SoC verification environment

- A means to check the full system before going to foundry
- A must have requirement for every SoC project
- Used to test the connections between different components of an SoC
- Check whether interrupts and their ISR routines are working correctly
- It not only helps in the exhaustive verification of the SoC for design and verification engineers but also helps the firmware and software engineers to write and debug the device drivers and application software

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Creating an SoC verification environment

Block diagram



AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Three steps to setup the environment

- Creating an SoC verification environment can be divided into three steps:

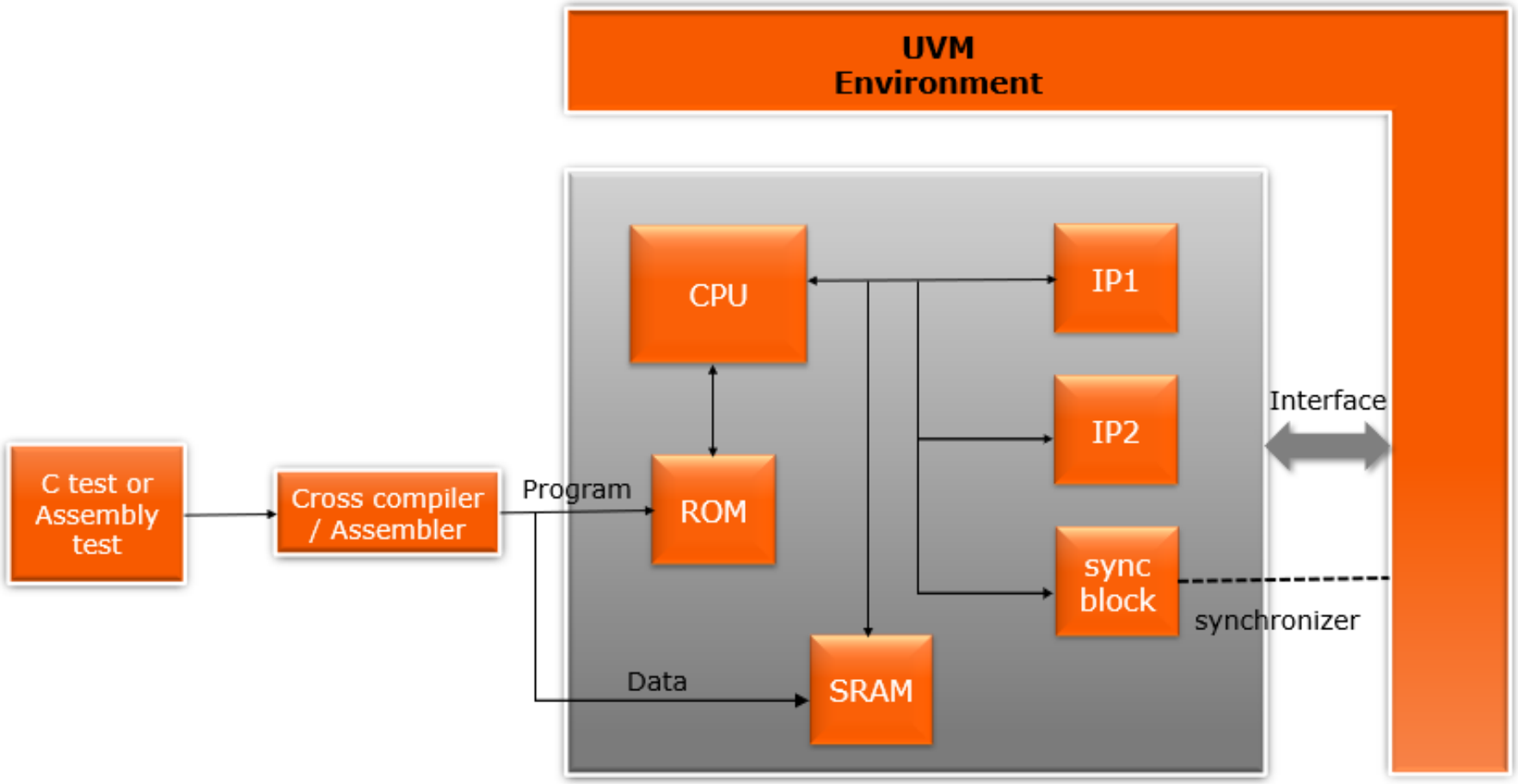
  1. UVM Architecture
  2. Converting C programs to binary files
  3. Synchronizing C programs and UVM tests

- We will be using a RISC-V based CPU and its associated tool chain

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

UVM Architecture

UVM architecture

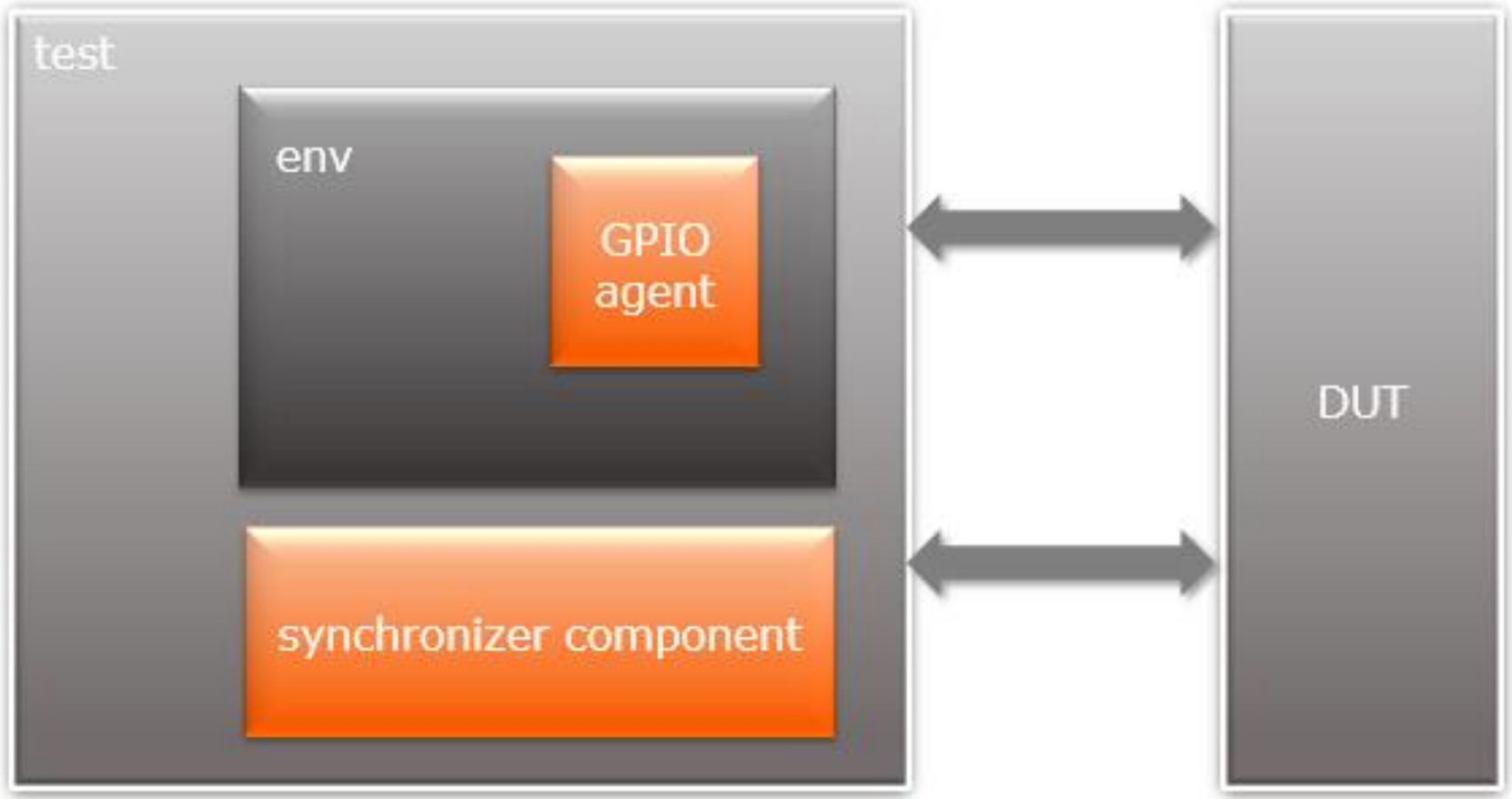


AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

UVM architecture

- Ability to write directed and random tests, which can test the various aspects of the SoC
- Raise/drop objection mechanism provides the flexibility to control the tests
- This feature is helpful in controlling the stop condition of the test from the C program
- Provide the flexibility of using powerful features of SystemVerilog such as assertions, functional coverage, randomization, etc.
- UVM agents written for various IPs (VIPs) can also be reused, for example, if certain pins of the SoC need driver for testing point of view

UVM architecture



AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

UVM architecture

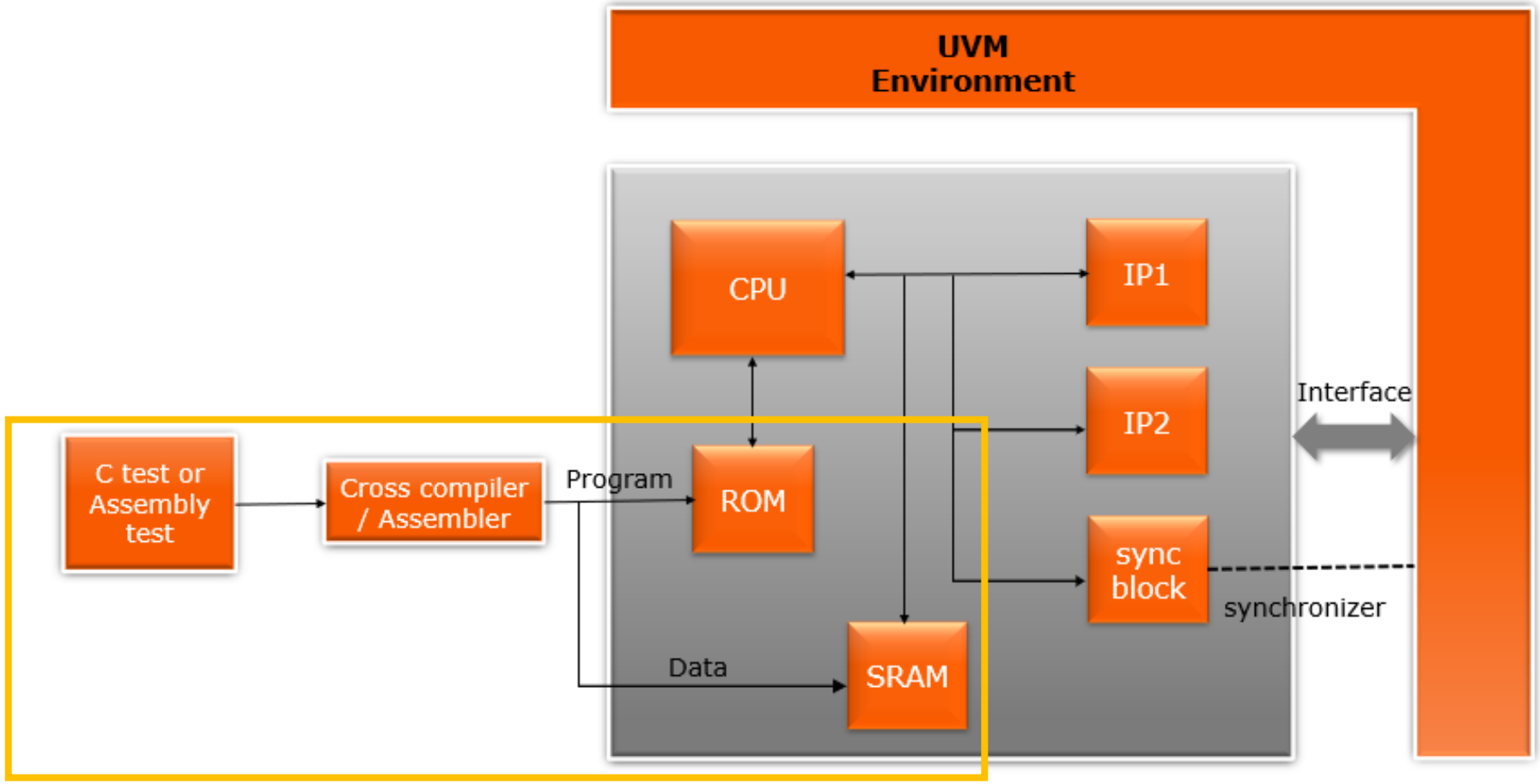
- For every UVM test, there will be a C test
- C tests are used to program the register blocks
- UVM tests are used to provide the start and stop condition of the tests and to drive the pins of an SoC, if required
- A shared space on the DUT will be present and will be accessed by the CPU through the frontdoor
- The synchronizer component in the testbench will access this location through the backdoor
- This shared space will be used for synchronization and for data transfer

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Converting C programs to binary files

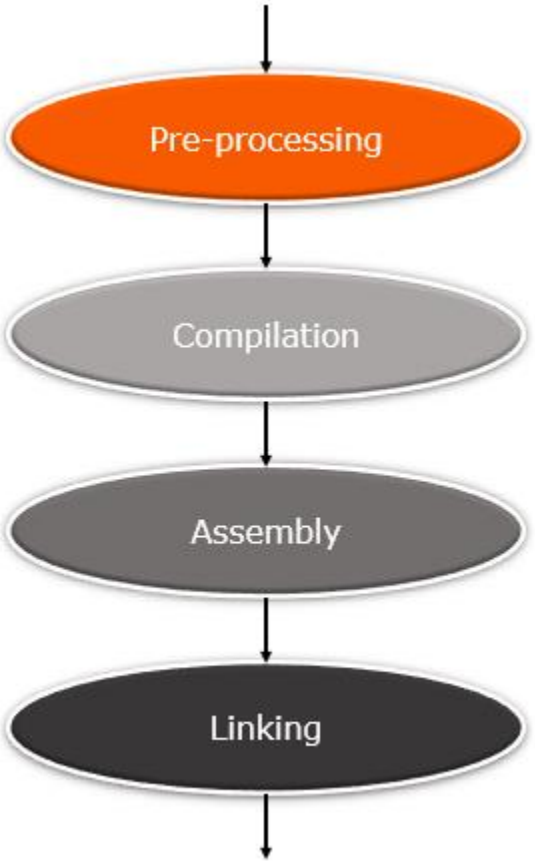
Converting C/assembly programs to binary files



AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Compiling C program : behind the scene

Compiling C program basically include four stages:



AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Pre-processing

- This is the first phase through which source code is passed
- This phase include:
  - Removal of comments
  - Expansion of macros
  - Expansion of the included files
  - Conditional compilation

Compilation

- In this stage, the preprocessed code is translated to assembly instructions specific to the target processor architecture

- Since our target processor is based on RISC-V ISA, we need the cross compiler for converting the code to RISC-V based assembly instructions

  “riscv32-unknown-elf-gcc ../start.S ../test.h ../gpio.h ../gpio.c -o gpio.o -static -nostdlib -nostartfiles -lm -lgcc -T ../link.ld”

- riscv32-unknown-elf-gcc is the command used for compiling the program targeted for embedded processors based on RISC-V 32bit instruction set

- This command comes along with various command line options of RISC-V toolchain

https://github.com/riscv/riscv-gnu-toolchain

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Compilation

- ‘test.h’, ‘gpio.h’ files contain the register information of the test block and GPIO block in the form of C structures. These structures are used for reading/writing to the block registers. They are automatically generated with the help of IDesignSpec™.

```
/**** This file is auto generated by IDesignSpec (http://www.agnisys.com) .
/* generated by       : Agnisys40*/
/* generated from     : C:\Users\Agnisys40\Desktop\test.docx*/
/* IDesignSpec rev    : idsbatch v6.26.38.0*/

#ifndef _TEST_REGS_H_
#define hwint32 uint32_t
#define hwint uint32_t
#define hwint8 uint8_t
#include <stdint.h>
#define _TEST_REGS_H_

/*  TEST DESCRIPTION : */
typedef union  {
    struct {
        hwint f : 32;           /* 31:0 SW=rw HW=na 0x0 */
    } bf;
    hwint  dw;
    hwint8 stride[0x8];

} test_test;

typedef struct {
    test_test  test[0xA];

} test_s;



#ifdef IDS_LITTLE_ENDIAN
    #define test_test_READMASK 0xFFFFFFFF
```

```
typedef struct {
    gpio_cfg  cfg;
    gpio_pin_cfg  pin_cfg[0x8];
    gpio_status  status;
    gpio_enable  enable;
    gpio_gpio_in  gpio_in;
    gpio_gpio_out  gpio_out;



} gpio_s;



#ifdef IDS_LITTLE_ENDIAN
    #define gpio_cfg_READMASK 0x3
    #define gpio_cfg_WRITEMASK 0x3
    #define gpio_cfg_VOLATILEMASK 0x0
    #define gpio_cfg_RESETMASK 0x3
    #define gpio_cfg_DEFAULT 0x00000000

    #define gpio_pin_cfg_READMASK 0x7F
    #define gpio_pin_cfg_WRITEMASK 0x7F
    #define gpio_pin_cfg_VOLATILEMASK 0x0
    #define gpio_pin_cfg_RESETMASK 0x7F
    #define gpio_pin_cfg_DEFAULT 0x00000000

    #define gpio_status_READMASK 0xFF
    #define gpio_status_WRITEMASK 0xFF
    #define gpio_status_VOLATILEMASK 0xFF
```

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Compilation

- ‘start.S’ file is a special file which we will talk about in later slides. ‘gpio.c’ file contains the code for initializing and testing the GPIO block.

```
int main(void) {

  int rd_data = 0x0;
  int data0 = 0x0;
  int data1 = 0x0;

  int data_array[2] = {0xf0,0x50} ;

  rd_data= test_h->test[0].dw;  //read start/stop test register

  while(rd_data == 0x0) {

    rd_data = test_h->test[0].dw;

  }

  data0 = test_h->test[5].dw;  ///data from the sv side   0x5
  data1 = test_h->test[6].dw;  //data from the sv side    0xff

  // configuring gpio pins as input
  gpio->pin_cfg[0].dw = 0x0;
  gpio->pin_cfg[1].dw = 0x0;
  gpio->pin_cfg[2].dw = 0x0;
  gpio->pin_cfg[3].dw = 0x0;

  // configuring gpio pins as output
  gpio->pin_cfg[4].dw = 0x1;
  gpio->pin_cfg[5].dw = 0x1;
  gpio->pin_cfg[6].dw = 0x1;
  gpio->pin_cfg[7].dw = 0x1;
```

```
  gpio->enable.dw = data1; // enable the interrupts
  gpio->cfg.dw = 0x3;      //enable the gpio block

  test_h->test[3].dw = 0x1;  //release the sv side

  gpio->gpio_out.dw = data_array[0];

  rd_data = gpio->gpio_in.dw;  //read the gpio in values 0x5

  test_h->test[8].dw = 0x1;
  test_h->test[9].dw = rd_data;

  if(rd_data != data0) {

    test_h->test[1].dw = 0x1;

   }

  rd_data = 0x0;
  rd_data = gpio->status.dw;  //read the status vlaue

  test_h->test[8].dw = 0x2;
  test_h->test[9].dw = rd_data;

  gpio->gpio_out.dw = data_array[1];

  block_h->enable.dw = 0x3;

  block_h->post.dw = 0x3;

  test_h->test[0].dw = 0x0;
   return 0;
}
```

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Assembly and linking

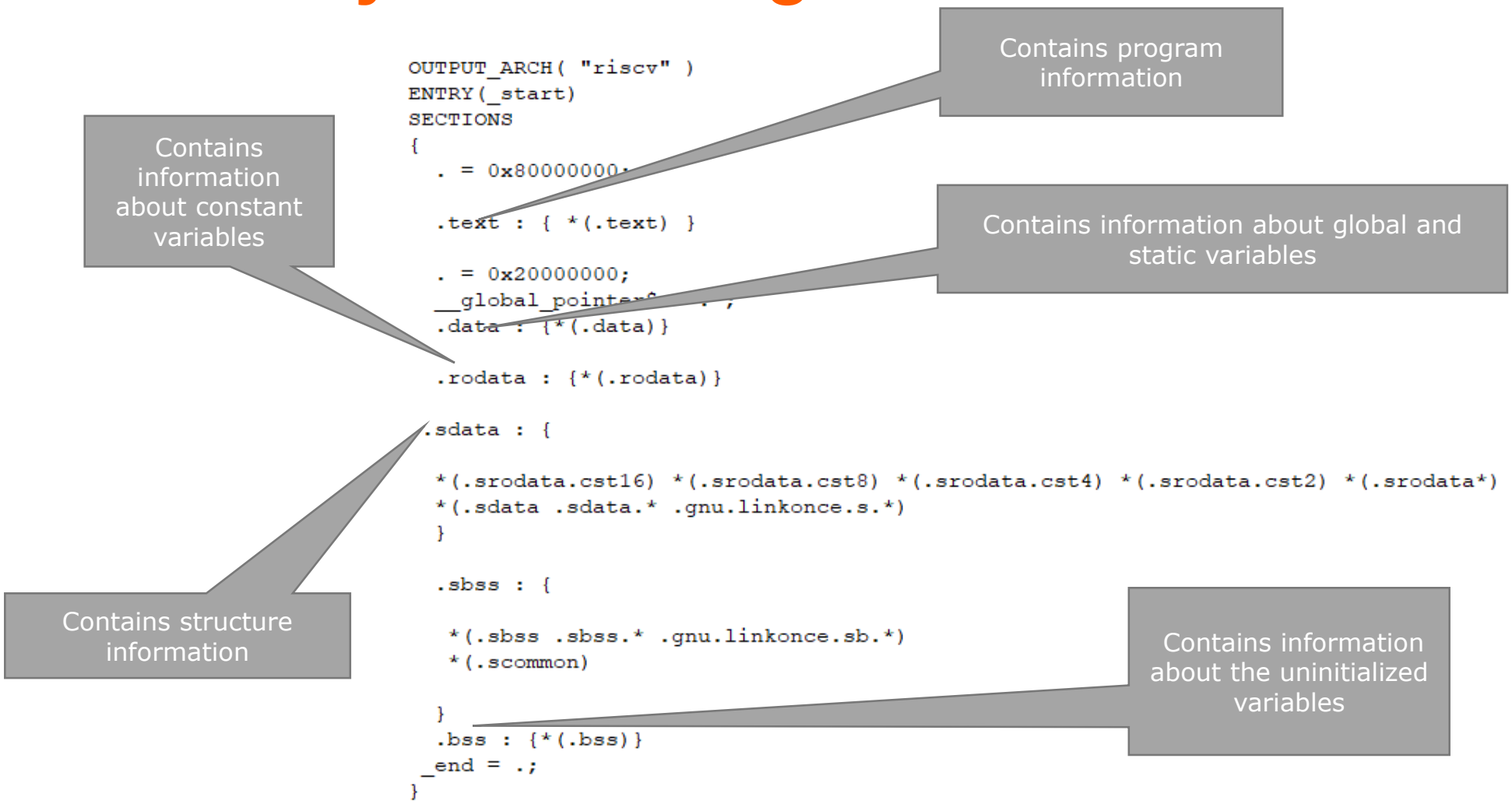
- In assembly stage, an assembler is used to translate the assembly instructions to object code. The output consists of actual instructions to be run by the target processor.

- In the linking stage, linker will arrange the pieces of object code so that functions in some pieces can successfully call functions in other ones. It will also add pieces containing the instructions for library functions used by the program.

- **“riscv32-unknown-elf-gcc ../start.S ../test.h ../gpio.h ../gpio.c -o gpio.o -static -nostdlib -nostartfiles -lm -lgcc -T ../link.ld”**

- ‘link.ld’ is the custom linker script that tells the linker in what memory locations to put different sections of the code.

Assembly and linking



AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Assembly and linking

- Following command dumps the elf file into human readable file:
  "riscv32-unknown-elf-objdump -Ds gpio.o > gpio.ds"

```
Disassembly of section .text:

80000000 <_start>:
80000000:   a04d                    j   800000a2 <startup>
80000002:   0001                    nop
80000004:   00000013            nop
80000008:   00000013            nop
8000000c:   00000013            nop

80000010 <trap_vector>:
80000010:   7119                    addi    sp,sp,-128
80000012:   c206                    sw  ra,4(sp)
80000014:   c40a                    sw  sp,8(sp)
80000016:   c60e                    sw  gp,12(sp)
80000018:   c812                    sw  tp,16(sp)
8000001a:   ca16                    sw  t0,20(sp)
8000001c:   cc1a                    sw  t1,24(sp)
8000001e:   ce1e                    sw  t2,28(sp)
80000020:   d022                    sw  s0,32(sp)
80000022:   d226                    sw  s1,36(sp)
80000024:   d42a                    sw  a0,40(sp)
80000026:   d62e                    sw  a1,44(sp)
80000028:   d832                    sw  a2,48(sp)
8000002a:   da36                    sw  a3,52(sp)
8000002c:   dc3a                    sw  a4,56(sp)
8000002e:   de3e                    sw  a5,60(sp)
80000030:   c0c2                    sw  a6,64(sp)
80000032:   c2c6                    sw  a7,68(sp)
80000034:   c4ca                    sw  s2,72(sp)
80000036:   c6ce                    sw  s3,76(sp)
80000038:   c8d2                    sw  s4,80(sp)
8000003a:   cad6                    sw  s5,84(sp)
```

```
8000057e <main>:
8000057e:   7179                    addi    sp,sp,-48
80000580:   d622                    sw  s0,44(sp)
80000582:   1800                    addi    s0,sp,48
80000584:   fe042623            sw  zero,-20(s0)
80000588:   fe042423            sw  zero,-24(s0)
8000058c:   fe042223            sw  zero,-28(s0)
80000590:   0f000793            li  a5,240
80000594:   fcf42e23            sw  a5,-36(s0)
80000598:   05000793            li  a5,80
8000059c:   fcf42023            sw  a5,-32(s0)
800005a0:   0001a783            lw  a5,0(gp) # 20000000 <test_h>
800005a4:   0007a783            lw  a5,0(a5)
800005a6:   fef42623            sw  a5,-20(s0)
800005aa:   a03d                    j   800005b6 <main+0x38>
800005ac:   0001a783            lw  a5,0(gp) # 20000000 <test_h>
800005b0:   0007a783            lw  a5,0(a5)
800005b2:   fef42623            sw  a5,-20(s0)
800005b6:   fec42783            lw  a5,-20(s0)
800005ba:   dbe5                    beqz    a5,800005ac <main+0x2e
800005bc:   0001a783            lw  a5,0(gp) # 20000000 <test_h>
800005c0:   579c                    lw  a5,40(a5)
800005c2:   fef42423            sw  a5,-24(s0)
800005c6:   0001a783            lw  a5,0(gp) # 20000000 <test_h>
800005ca:   5b9c                    lw  a5,48(a5)
800005cc:   fef42223            sw  a5,-28(s0)
800005d0:   0081a783            lw  a5,8(gp) # 20000008 <gpio>
800005d4:   0007a223            sw  zero,4(a5)
800005d8:   0081a783            lw  a5,8(gp) # 20000008 <gpio>
800005dc:   0007a423            sw  zero,8(a5)
800005e0:   0081a783            lw  a5,8(gp) # 20000008 <gpio>
800005e4:   0007a623            sw  zero,12(a5)
```

Equivalent assembly instruction

Data stored at that location

Physical address

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

start.S file

start.S file

- ‘start.S’ file contains the entry point of the program
- This file contains assembly instructions for startup code and these instructions are run first prior to the programs
- It includes initialization of all general-purpose registers to zero values
- It includes initialization of the stack pointer
  - Stack pointer initialization is a must requirement if you want to run C programs
  - Stack pointer is used to store the intermediate results of the functions, state of the program when an interrupt occurs, and state of the function when a function calls another function
- It also includes instructions to initialize certain embedded registers of the processor for enabling different functionalities

start.S file

```
.text
.global _start
_start:
   j startup

.align 4

trap_vector: ///only for the swerve Core

   addi sp, sp, -32*REGBYTES

   sw x1, 1*REGBYTES(sp)
   sw x2, 2*REGBYTES(sp)
   sw x3, 3*REGBYTES(sp)
   sw x4, 4*REGBYTES(sp)
   sw x5, 5*REGBYTES(sp)
   sw x6, 6*REGBYTES(sp)
   sw x7, 7*REGBYTES(sp)
   sw x8, 8*REGBYTES(sp)
   sw x9, 9*REGBYTES(sp)
   sw x10, 10*REGBYTES(sp)
   sw x11, 11*REGBYTES(sp)
   sw x12, 12*REGBYTES(sp)
   sw x13, 13*REGBYTES(sp)
   sw x14, 14*REGBYTES(sp)
   sw x15, 15*REGBYTES(sp)
   sw x16, 16*REGBYTES(sp)
   sw x17, 17*REGBYTES(sp)
   sw x18, 18*REGBYTES(sp)
```

```
startup:
   li x1,0
   li x2,0
   li x3,0
   li x4,0
   li x5,0
   li x6,0
   li x7,0
   li x8,0
   li x9,0
   li x10,0


la t0, trap_vector
csrw mtvec, t0

.option push
.option norelax
la gp, __global_pointer$
.option pop
li sp, 0x20004000

call _init
call main
```

```
void _init(){

  int rdata;
  unsigned int data;

  ////////////////////////////mstatus////
  asm("csrr %0, 0x300" : "=r"(rdata) :);////
  rdata = rdata | 0x00000008;
  // mstatus global interrupt enable
  asm("csrw 0x300,%0" : : "r"(rdata) :);

  ////////////////////////////mie/////////
  data = 1<<11;
  //write to mie swerve core reigster for enabling external interrupt
  asm("csrw 0x304,%0" : : "r"(data) :);

  pic_init();

}
```

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Loader

- A **loader** is a system program, which takes the object code of a program as input and prepares it for execution
- A **loader** loads the machine code corresponding to the object modules into the allocated memory space and makes the program ready to execute
- Program data extracted from the ‘.text’ section of object code goes into the instruction memory and the data used by the program goes into the data memory
- Since we do not have any kind of loader available in the SoC verification environment, we need to load the binary data into the memories so that processor can process it
- Verilog provides the $readmemh and $readmemb functions for the same
- These allow us to initialize memory from a text file with either hex or binary values

Loader

“riscv32-unknown-elf-objcopy -O verilog --only-section ".text*" --set-start 0x0 gpio.o program.hex”

- riscv32-unknown-elf-objcopy is a command line option used to extract the binary information of the .text section from the object file and dump into the program.hex file
- program.hex file will contain the program bitstream that will be executed by the processor

“riscv32-unknown-elf-objcopy -O verilog --only-section ".data*" --only-section ".rodata*" --only-section ".sdata*" --only-section ".bss*" --set-start 0x0 gpio.o data.hex”

- extract the information from the .data, .rodata, .sdata and .bss sections and dump it into the data.hex file
- Data.hex file contains data that the program will use during its execution

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

How to setup external interrupts

How to setup external interrupts

Why do we need interrupts?

- Interrupts are used to tell the processor that a device requires attention
- Interrupts are used to interrupt the active process, store the process state, service the device, and continue with the running program as if nothing had happened
- Interrupts are the fundamental building blocks of multi-tasking operating systems
- Interrupts ensure that devices are serviced in a timely manner

How to setup external interrupts

- In order to setup the interrupts, first we must enable the interrupts' functionality in processor

- We have used the SweRV_EH1 Core based on RISC-V architecture in our design

- So in order to enable the external interrupts, first we must set the mie bit of the mstatus register. This bit is used to enable/disable all interrupts whether they are internal, external, or timer interrupts

- The mie register than controls enabling/disabling of internal/machine timer interrupts, external interrupts, software interrupts, and local error interrupts. The meie bit of the mie register is used to control the external interrupts, so, we have to set the mie bit of mstatus and the meie bit of the mie register in order to enable the external interrupts.

- After receiving the interrupt request, the processor will finish its current execution and then jump to service the interrupt request

How to setup external interrupts

- Processor must know the address from which it will fetch the instructions for servicing the interrupt
- In RISC-V architecture, the mtvec register contains the address of the routine
- This register needs to be initialized with the address where the interrupt service routines are stored

https://github.com/chipsalliance/Cores-SweRV

How to setup external interrupts

```
trap_vector:

   addi sp, sp, -32*REGBYTES     la t0, trap_vector
                                 csrw mtvec, t0
   sw x1, 1*REGBYTES(sp)
   sw x2, 2*REGBYTES(sp)         .option push
   sw x3, 3*REGBYTES(sp)         .option norelax
   sw x4, 4*REGBYTES(sp)         la gp, __global_pointer$
   sw x5, 5*REGBYTES(sp)         .option pop
   sw x6, 6*REGBYTES(sp)         li sp, 0x20004000
   sw x7, 7*REGBYTES(sp)
   sw x8, 8*REGBYTES(sp)         call _init
   sw x9, 9*REGBYTES(sp)         call main
   sw x10, 10*REGBYTES(sp)
 // meicpct Capture winning claim id and priority
 csrwi 0xBCA, 1

 call interrupt_handler

 lw x1, 1*REGBYTES(sp)
 lw x2, 2*REGBYTES(sp)
 lw x3, 3*REGBYTES(sp)
 lw x4, 4*REGBYTES(sp)
 lw x5, 5*REGBYTES(sp)
 lw x6, 6*REGBYTES(sp)
 lw x7, 7*REGBYTES(sp)
 lw x8, 8*REGBYTES(sp)
 lw x9, 9*REGBYTES(sp)
 lw x10, 10*REGBYTES(sp)
```

```
void _init(){

  int rdata;
  unsigned int data;

  ////////////////////////////mstatus////
  asm("csrr %0, 0x300" : "=r"(rdata) :);////
  rdata = rdata | 0x00000008;
  // mstatus global interrupt enable
  asm("csrw 0x300,%0" : : "r"(rdata) :);

  ////////////////////////////mie/////////
  data = 1<<11;
  //write to mie swerve core reigster for enabling external interrupt
  asm("csrw 0x304,%0" : : "r"(data) :);

  pic_init();

}
```

```
void interrupt_handler(){

    unsigned int idx;
    void(*func_ptr[])() = {dummy_handle,gpio_handle,intr_handle};
    asm("csrr %0, 0xFC8" : "=r"(idx) :);

    idx = (idx >> 2) & 0x000000ff;

    (*func_ptr[idx])();

}
```

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

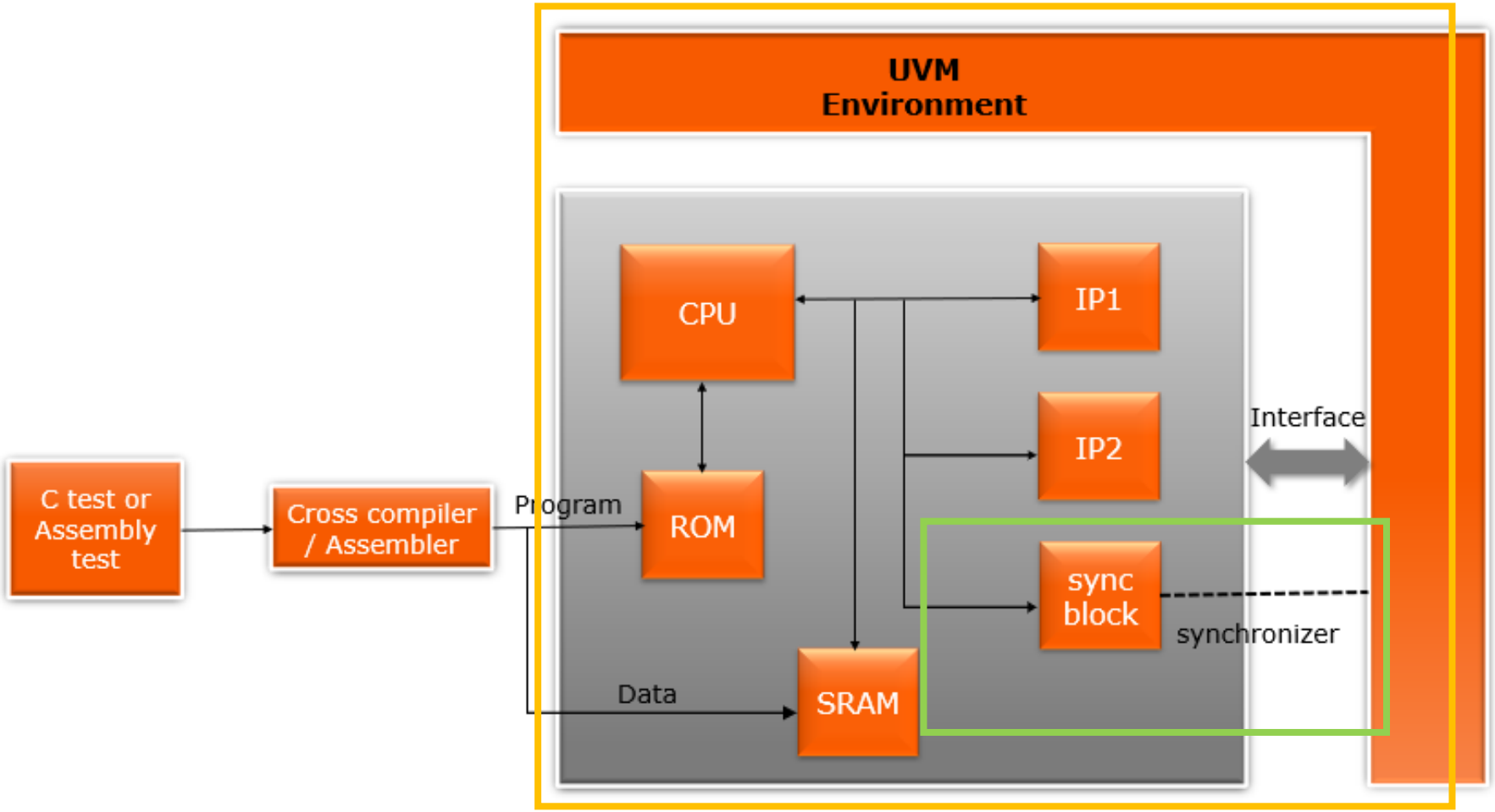
AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Synchronizing C programs and UVM tests

Why do we need synchronization b/w C programs and UVM tests?

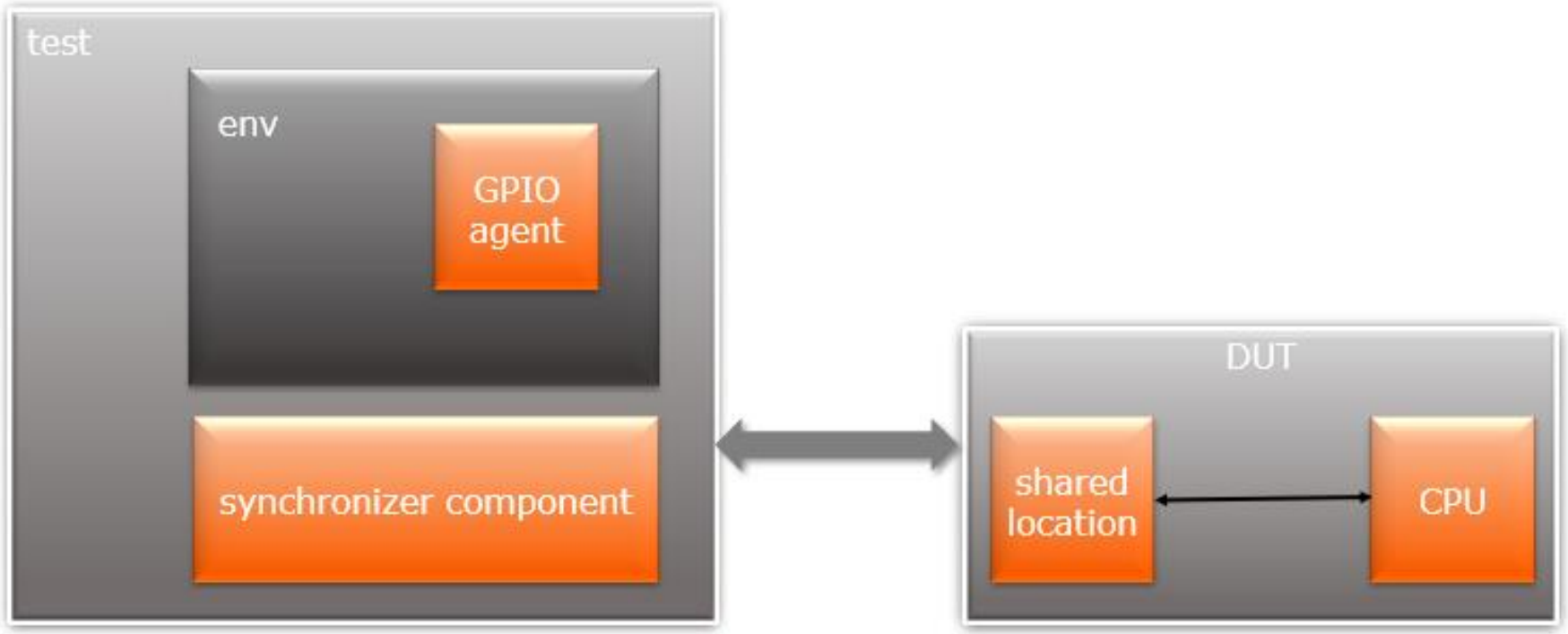
- C and UVM are two different worlds
- We need certain synchronization so that both programs can interact with each other
- We may want to drive some pins of an SoC after certain blocks are configured for testing some features
- We may want in the C program to stop for some events that are triggered by the UVM side
- There may be requirements for generating errors for testing purpose from the C program
- We may want to finish the test when the C program has completed its execution

Synchronizing C programs and UVM tests



AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Synchronizing C programs and UVM tests



AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Rules to synchronize between C and UVM tests

- Certain rules are defined for synchronization and passing information between C and UVM tests
- A certain space in the memory or in the block will be used for sharing the information between C and UVM tests
- Memory space of SRAM or certain IP register can be used for it and this shared space should not be used by the program and it should only be used for synchronization purpose
- C programs access this shared storage through the frontdoor (time consuming) whereas the UVM side accesses this space through the backdoor
- For our environment we have used 10 32-bit registers of the synchronizer block
- First 5 registers are used for the synchronization and rest are used for passing information between the two sides

Rules to synchronize between C and UVM tests

Location	Functionality
Location_0	Provides starts or stops condition to uvm_test
Location_1	Provides uvm_error functionality to C program
Location_2	Provides uvm_fatal functionality to C program
Location_3	Provides waiting event on UVM side and event trigger condition on C side
Location_4	Provides waiting event on C side and event trigger condition on UVM side
Location_5 to Location_9	Used to pass data between two sides

Rules to synchronize between C and UVM tests

Location_0 :-

The 0th bit of location_0 gives information about the start and stop of the test. A transition from 0->1 represent the start condition of the test, whereas the transition from 1->0 represent the stop condition of the test. Both C program and UVM side can start the test. The user can use the stop functionality from C program to call the drop objection of the test or may use other ways to stop the test.

Location_1 :-

This location provides C program a functionality to generate the UVM_ERROR if an error condition occurs. Whenever a C program writes 1 to that location, a uvm_error condition is generated, and a counter is incremented on the UVM side to display the number of errors that have occurred. Whenever C writes 1 to that location, after giving an error the synchronizer will clear this register from the backdoor in order to take another error from the C program.

Rules to synchronize between C and UVM tests

Location_2 :-

This location provides C program a functionality of generating a UVM_FATAL condition. Whenever C program writes 1 to this location, UVM will generate a fatal condition and the test will be automatically stopped.

Location_3 :-

This location provides an event for the UVM side. UVM side can use this as waiting condition after which it can resume its operation. This event is triggered by C program by writing 1 to this location. After the event is triggered, this location will be cleared by the UVM side for next synchronization. A typical example of this synchronization would be, if you want to drive some pins of the SoC but only after certain blocks are initialized.

Location_4 :-

This location provides an event for C program. The C program can wait on this event which will be triggered by the UVM side. C program continuously reads this location until its value become 1 and then signals the UVM side by writing 1 to location_3.

Rules to synchronize between C and UVM tests

Location_5 to Location_9 :-

Location_0 to Location_4 are used for synchronization purpose whereas Location_5 to Location_9 are used for passing data from C program to UVM or vice versa. C program can use this location for getting the randomized data from UVM side.

UVM	C
sync.wait_for_start()	test_h->test[0].dw = 0x1
sync.wait_for_stop()	test_h->test[0].dw = 0x0
sync.test_start()	While(test_h->test[0].dw==0) {}
sync.test_stop()	While(test_h->test[0].dw==1) {}
sync.check_error() generate UVM_ERROR, increment the static counter and after that clear that location.	test_h->test[1].dw = 0x1
sync.check_fatal() generate UVM_FATAL condition	test_h->test[2].dw = 0x1
sync.wait_for_cpu()	test_h->test[3].dw = 0x1
sync.release_cpu()	While(test_h->test[4].dw ==0){} test_h->test[3].dw=0x1
sync.wr_data(location , data)	data0 = test_h->test[location].dw
sync.rd_data(location, data)	test_h->test[location].dw = data

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Summary

Summary

- SoC verification environment is critical for every project
- It helps thorough testing of the entire system, with all the original components that will be present in a chip
- Same approach can be used to verify other SoCs whose processors are based on other ISAs
- Only change will be conversion of C programs and the interrupt setup
- Rest of the steps will remain same
- This environment has one disadvantage: long and time-consuming simulations because of the presence of CPU along with the IPs
- Our tool – ARVV – helps in creating this environment

References

- https://github.com/chipsalliance/Cores-SweRV/tree/master/docs
- https://github.com/riscv/riscv-gnu-toolchain
- https://sourceware.org/binutils/docs-2.27/ld/Scripts.html
- http://ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html
- https://embeddedartistry.com/blog/2019/04/08/a-general-overview-of-what-happens-before-main/
- https://github.com/riscv/riscv-4th-workshop-tutorials/
- https://riscv.org/specifications/privileged-isa/

Agnisys Webinar Series

BRINGING THE LATEST AUTOMATION IN IP/FPGA/SOC TO YOUR HOME!

Time : 10:00 AM – 11:00 AM PDT

08-April-2020 Correct by construction SV UVM code with DVinsight - a smart editor	30-April-2020 Advanced UVM RAL - callbacks, auto-mirroring, coverage model, and more	28-May-2020 Steps to setup RISC-V based SOC Verification Environment
09-April-2020 Creating portable UVM sequences with ISequenceSpec	7-May-2020 Functional safety and security in embedded systems	04-June-2020 Automatic verification using Specta -AV - a boost to verification productivity
16-April-2020 Register automation from SystemRDL to PSS - Basic to Pro	14-May-2020 IP generators - the next wave of design creation	11-June-2020 AI based sequence detection for verification and validation of IP/SoCs
23-April-2020 Cross platform specification to code generation for IP/SoC with IDS-NG	21-May-2020 A flexible and customizable flow for IP connectivity and SoC design assembly	18-June-2020 Understanding clock domain crossings

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

About Agnisys

- The EDA leader in solving complex design and verification problems associated with HW/SW interface
- Formed in 2007
- Privately held and profitable
- Headquarters in Boston, MA
  - ~1000 users worldwide
  - ~50 customer companies
- Customer retention rate ~90%
- R&D centers (US and India)
- Support centers - committed to ensure comprehensive support
  - Email : support@agnisys.com
  - Phone : 1-855-VERIFYY
  - Response time within one day; within hours in many cases
  - Multiple time zones (Boston MA, San Jose CA and Noida India)

IDESIGNSPEC™ (IDS) EXECUTABLE REGISTER SPECIFICATION

Automatically generate UVM models, Verilog/VHDL models, coverage model, software model, etc.

AUTOMATIC REGISTER VERIFICATION (ARV)

ARV-Sim™ : Create UVM test environment, sequences, and verification plans, and instantly know the status of the verification project

ARV-Formal™ : Create formal properties and assertions, and coverage model from the specification

ISEQUENCESPEC™ (ISS)

Create UVM sequences and firmware routines from the specification

DVinsight™ (DVi)

Smart editor for SystemVerilog and UVM projects

IDS – Next Generation™ (IDS-NG)

Comprehensive SoC/IP spec creation and code generation tool

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

THANK YOU