AXI bridge documentation

August 14, 2015

Contents

List of Tables

List of Figures

Introduction

This document describes the AXI bridge of the SecBus project. Chapter 1 is the user guide and presents the bridge on a purely functional point of view. Chapter 2 gives examples of use of the bridge on the ZedBoard[2], a prototyping board based on the Xilinx Zynq core[1].

Revision history

Date	**Version**	**Revision**
2015-07-27	1.0	Initial release.

Table 1: Revision history

Chapter 1

User guide

Introduction

This chapter presents the functional view of the AXI bridge. The reader interested in using the AXI bridge from a pure functional perspective will find the description of the internal registers and their role.

1.1 Interface

The AXI bridge is a hardware component with a system clock and reset, two slave AXI ports (`S0_AXI` and `S1_AXI`), one master AXI port (`M_AXI`), an 8-bits general purpose input (`GPI`), an 8 bits general purpose output (`GPO`) and a active high, synchronous, reset for its internal registers (`SRST`). Table 1.1 lists the input and output ports.

Name	Direction	Bit-width	Description
`ACLK`	input	1	System clock
`ARESETN`	input	1	System reset
`SRST`	input	1	Registers synchronous reset
`S0_AXI`	-	-	AXI lite slave interface
`S1_AXI`	-	-	AXI slave interface
`M_AXI`	-	-	AXI master interface
`GPI`	input	8	General Purpose Input
`GPO`	output	8	General Purpose Output

Table 1.1: AXI bridge input and output ports

The bridge can be customized with 3 generic parameters. Table 1.2 lists the generic parameters. Their role and description is given in the following sections.

Name	Type	Default
S1_AXI_ADDRESS_AND_MASK	32-bits vector	0x3fff_ffff
S1_AXI_ADDRESS_OR_MASK	32-bits vector	0x0000_0000
S0_AXI_ADDRESS_WIDTH	Integer	12

Table 1.2: AXI bridge generic parameters

1.2 Internal registers

The bridge contains 21 32-bits internal registers. When SRST is asserted high or when the system reset is asserted low, the internal registers are reset to their default value. The difference between the two resets is that the system reset also resets other registers (state registers of state machines...).

The S0_AXI port is used to access the internal registers. The S0_AXI_ADDRESS_WIDTH generic parameter defines the bit-width of the S0_AXI read and write addresses. Accessing an unmapped address with S0_AXI returns a DECERR AXI response.

Some internal registers are read-write and some are read-only. Writing a read-only register returns a SLVERR AXI response. Some registers have reserved bits. They read as zeroes and writing them has no effect.

The address map (relative to the base address of the S0_AXI port in the host system), read-write attribute and short description of the internal registers is given in table 1.3.

Table 1.3: AXI Bridge registers table

Name	**Address**	**Dir.**	**Description**
gpir	0x40000000	r	Current value of GPI input (8 LSBs only)
gpor	0x40000004	rw	Value sent on General Purpose Output when GPI=0x01 (8 LSBs only)
msk	0x40000008	rw	Used to compute the one-bit activity indicators from the AXI transaction counters
aw	0x4000000c	r	Counts the completed transactions on the Address Write AXI channel
ar	0x40000010	r	Counts the completed transactions on the Address Read AXI channel
w	0x40000014	r	Counts the completed transactions on the Write data AXI channel
r	0x40000018	r	Counts the completed transactions on the Read data AXI channel
b	0x4000001c	r	Counts the completed transactions on the Write response AXI channel
cfg	0x40000020	rw	Configuration and status register
before_r	0x40000024	r	Last rdata value before trigger
after_w	0x40000028	r	First wdata value after trigger
after_r	0x4000002c	r	First rdata value after trigger
rtrig	0x40000030	rw	First read data value of trigger pattern
rtrig2	0x40000034	rw	Second read data value of trigger pattern

Name	Address	Dir.	Description
`rtrig3`	0x40000038	rw	Third read data value of trigger pattern
`wtrig`	0x4000003c	rw	First write data value of trigger pattern
`wtrig2`	0x40000040	rw	Second write data value of trigger pattern
`wtrig3`	0x40000044	rw	Third write data value of trigger pattern
`ival`	0x40000048	rw	Value to inject
`iread`	0x4000004c	rw	The read value to overwrite during injection
`fifo`	0x40000050	r	FIFO read register

1.3 Functional description

The bridge forwards the AXI requests it receives on the `S1_AXI` port to the `M_AXI` port and forwards the responses received on the `M_AXI` port to the `S1_AXI` port. An address transform is applied to the `S1_AXI` read and write requests: the 32 bits addresses are bitwise AND-masked with the `S1_AXI_ADDRESS_AND_MASK` 32 bits generic parameter and then bitwise OR-masked with the `S1_AXI_ADDRESS_OR_MASK` 32 bits generic parameter:

```
M_AXI.AWADDR <= S1_AXI_ADDRESS_OR_MASK or
  (S0_AXI.AWADDR and S1_AXI_ADDRESS_AND_MASK);
M_AXI.ARADDR <= S1_AXI_ADDRESS_OR_MASK or
  (S0_AXI.ARADDR and S1_AXI_ADDRESS_AND_MASK);
```

The `AW`, `AR`, `W`, `R` and `B` registers are counters. They count the number of completed transactions on the five AXI channels of the `S1_AXI` to `M_AXI` path. The value of `MSK` is used to condense the counter values into 5 single bit indicators (`AWI`, `ARI`, `WI`, `RI` and `BI`) by a AND-masking followed by a OR-reduction:

```
AWI <= or_reduce(MSK and AW);
ARI <= or_reduce(MSK and AR);
WI  <= or_reduce(MSK and W);
RI  <= or_reduce(MSK and R);
BI  <= or_reduce(MSK and B);
```

The bridge offers two more features than simple forwarding of AXI requests and responses:

- Capture in a FIFO the data read by the CPU on the `S1_AXI` port,
- Injection of a forget data on the `S1_AXI` port to replace a data read by the CPU.

These two features can be used to demonstrate the effect of attacks against the content of external memories (memory readout, memory bus sniffing, memory overwriting, memory bus injection...). The capture and the injection are triggered by two programmable sub-triggers: a read sub-trigger and a write sub-trigger. Each sub-trigger compare the consecutive read (written) values on the `S1_AXI` port with a sequence of reference values. The length of each reference sequence is programmable from 0 (sub-trigger activated by default) to 3 (sub-trigger activated after 3 consecutive value matches). When a comparison fails, the corresponding sub-trigger is reset and the matching process restarts from the beginning. The global trigger is fired only when the two sub-triggers are fired. The reference sequence lengths are defined by the

CFG.NUMR and CFG.NUMW fields of the CFG register. The sequences of reference values are defined in the RTRIG, RTRIG2 and RTRIG3 registers for the read sub-trigger and in the WTRIG, WTRIG2 and WTRIG3 registers for the write sub-trigger. The CFG.TEN, CFG.CEN and CFG.IEN flags are used to enable or disable the trigger, capture and injection, respectively. They are automatically de-asserted when the action they are enabling is done. A new trigger, capture and / or injection can be programmed by preparing a new trigger condition (CFG.NUMR, CFG.NUMW, RTRIGx, WTRIGx) and re-enabling the trigger, capture and / or injection.

Captured read data are stored in a 20-words FIFO. The capture stops when the FIFO is full. The content of the FIFO can be retrieved by reading the FIFO register. The current status of the FIFO is given by the CFG.FFULL and CFG.FEMPTY read-only flags.

Injection takes place when the trigger has been fired, a data is read from the S1_AXI port and the read value matches the value stored in the IREAD register. The read value is then replaced with the value stored in the IVAL register.

When set, the CFG.RST soft reset disables the trigger, capture and injection and resets the capture FIFO to empty. CFG.RST is de-asserted automatically on the next clock cycle.

Three more read-only registers (BEFORE_R, AFTER_W and AFTER_R) store the last read value before the trigger is fired, the first written and read values after the trigger is fired, respectively. They are read-only.

The least significant byte of GPIR always contains the current value of the GPI primary input. Its value selects the value sent to the GPO primary output, as listed in table 1.4 (where FFULL, FEMPTY, RSEQ, WSAQ, TEN, CEN and IEN are the fields with same names in the CFG configuration register).

GPIR[7...0]	GPO source
0x00	GPOR[7...0]
0x01	FFULL|FEMPTY|0|AWI|ARI|WI|RI|BI
0x02	RSEQ|WSEQ|0|TEN|CEN|IEN
other	"01010101" (0x55)

Table 1.4: Value of GPO as a function of GPIR

Figure 1.1 represents the bridge.

1.4 Layout of the internal registers

The layout of the GPIR, GPOR and CFG registers is depicted in figures 1.2, 1.3 and 1.4. The layout of the other registers is trivial.

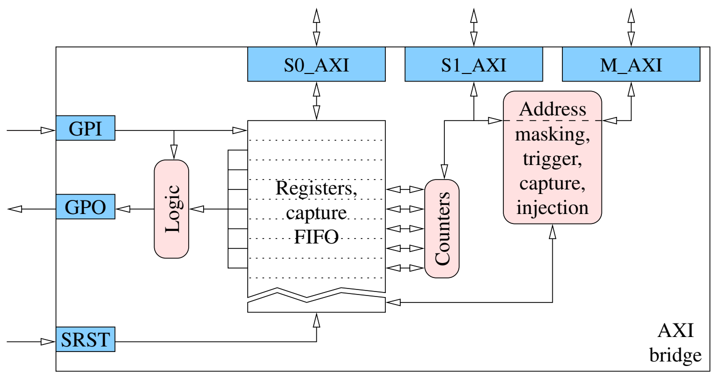


Figure 1.1: AXI bridge

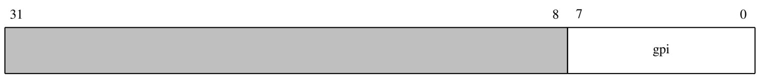


Figure 1.2: AXI Bridge `gpir` register layout: General Purpose Input Register

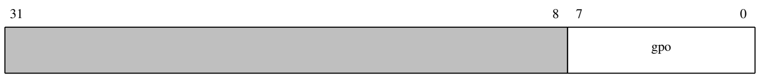


Figure 1.3: AXI Bridge `gpor` register layout: General Purpose Output Register

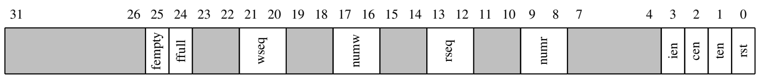


Figure 1.4: AXI Bridge `cfg` register layout: ConFiGuration

Chapter 2

Application notes

2.1 Introduction

This application note proposes examples of use of the bridge on the ZedBoard[2], a prototyping board based on the Xilinx Zynq core[1]. The AXI bridge is integrated in the Programmable Logic (PL) of the Zynq core and, just like the AXI simple bridge, creates an Alternate Address Space (AAS) for the ARM processor of the Processing System (PS) of the Zynq by routing its read-write requests to the DDR memory back and forth. The capture and injection capabilities of the bridge are used to demonstrate the effect of attacks against the content of external memories (memory readout, memory bus sniffing, memory overwriting, memory bus injection...).

Figure 2.1 represents the bridge in its environment.

The 4GB address space of the ARM processor is split in four 1GB sub-spaces. The bridge makes use of 3 of them:

- [0GB..1GB[: Regular Address Space (RAS). The direct access to the DDR controller falls in RAS.

- [1GB..2GB[: Control Address Space (CAS). Used by the processor to access the internal registers of the bridge. All processor requests in CAS are sent to AXI master port M_AXI_GP0 of the PS which is connected to the AXI slave port S0_AXI of the bridge in the PL.

- [2GB..3GB[: Alternate Address Space (AAS). Used by the processor to access the DDR controller through the bridge in the PL. All processor requests in AAS are sent to AXI master port M_AXI_GP1 of the PS which is connected to the AXI slave port S1_AXI of the bridge in the PL.

The bridge forwards the AXI requests received from its S1_AXI AXI slave port to its M_AXI AXI master port, with a 2GB address down-shift that brings the addresses back in the RAS. This is done automatically when using the default values of the S1_AXI_ADDRESS_AND_MASK and S1_AXI_ADDRESS_OR_MASK generic parameters. The M_AXI port is connected to the S_AXI_HP0 slave port of the PS. This way, when the processor accesses a memory location in the AAS, it is exactly like if it was addressing the equivalent location in RAS, except that the AXI transactions flow through the bridge in the PL.

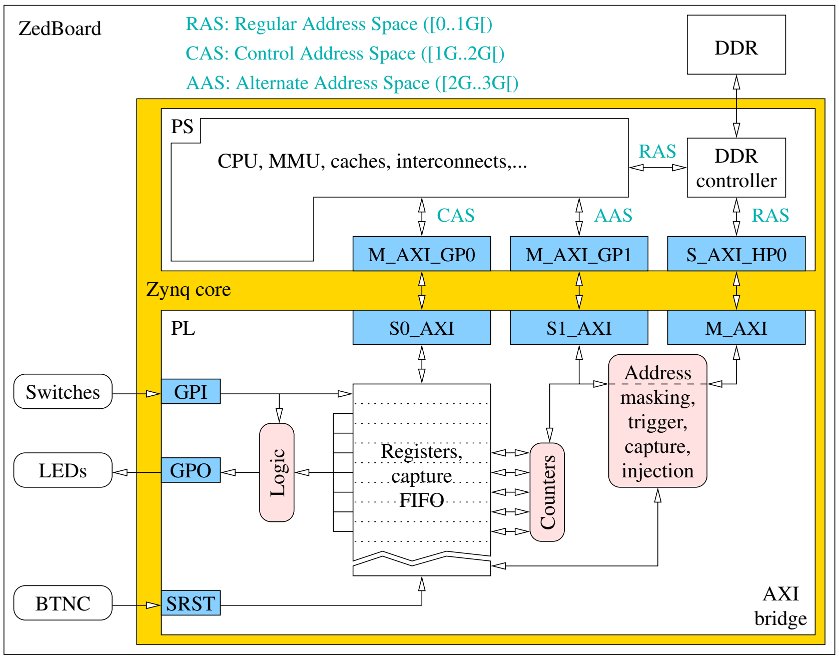

ZedBoard
RAS: Regular Address Space ([0..1G[)
CAS: Control Address Space ([1G..2G[)
AAS: Alternate Address Space ([2G..3G[)
DDR
PS
CPU, MMU, caches, interconnects,...
RAS
DDR controller
CAS
AAS
RAS
M_AXI_GP0
M_AXI_GP1
S_AXI_HP0
Zynq core
PL
S0_AXI
S1_AXI
M_AXI
Switches
GPI
Address masking, trigger, capture, injection
LEDs
GPO
Logic
Registers, capture FIFO
Counters
BTNC
SRST
AXI bridge

Figure 2.1: AXI bridge in ZedBoard

The `GPI` primary input is connected to the 8 switches of the ZedBoard, the `GPO` primary output is connected to the 8 user LEDs and the `SRST` synchronous reset of the internal registers is connected to the centre button of the 5 press buttons pad. So, pressing the centre button of the 5 buttons pad of the ZedBoard resets the internal registers.

Thanks to this, the `AR`, `AW`, `W`, `R` and `B` AXI activity indicators drive the LEDs in the `GPIR=0x01` mode and it is thus possible to observe the `S1_AXI` to `M_AXI` activity. Writting 8 to the `MSK` register, for instance, will force the LEDs to blink every 8 transactions when the switches (`GPI`) are in the `0x01` configuration. Of course, if the `MSK` register is left to its reset value (0), no activity will be visible.

Important note: thanks to the bridge the complete 1GB `RAS` is also mapped to a 1GB `AAS`. On Zynq-based boards that do not map this entire `RAS` GB (like, for instance, the ZedBoard with only 512 MB of DDR), the unmapped region in `RAS` has a corresponding unmapped region in `AAS`. Accessing one or the other will lead to unexpected results (aliasing or errors). Moreover, depending on the configuration, some Zynq systems have a reserved low addresses range that cannot be accessed from the `S_AXI_HP` ports. In these systems this low range can be accessed in the `RAS` but not in the `AAS`.

2.2 Configuring the ZedBoard with the bridge

The provided SD card archive contains all files needed to run a `busybox` on top of the Linux kernel with all memory accesses routed to the AXI bridge in the PL. The device tree blob (and sources), the Linux kernel image and the U-Boot environment variables definitions are customized such that the available DDR memory seen by the Linux kernel is entirely in the AAS and limited to the `[0x8800_0000, 0xa000_0000[` range. The first 128 MB (`[0x8000_0000, 0x8800_0000[`) is excluded for good technical reasons that are out of the scope of this note.

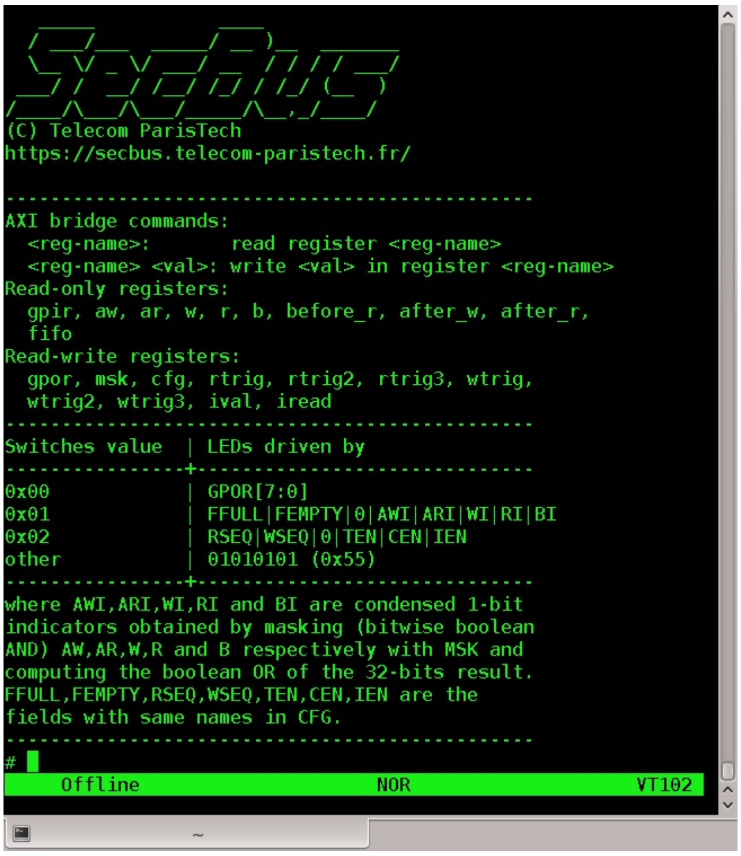
The SD card archive content is:

```
.bashrc ................ Initialization script
                         (aliases definitions)
boot.bin ............... Zynq boot image (FSBL,
                         bitstream and U-Boot)
COPYING ................ License (English)
COPYING-FR ............. Licence (French)
devicetree.dtb ......... Linux device tree blob
doc/
  axi_bridge.pdf ........ This document
  bitstream.bit ......... Bitstream
  dts/ .................. Device tree sources
    device-tree.mss
    pl.dtsi
    skeleton.dtsi
    system.dts .......... Default device tree
    system.dts.edited ... Modified device tree
    zynq-7000.dtsi
  fsbl.elf .............. First Stage Boot Loader ELF
  system.sysdef ......... System definition
  u-boot.elf ............ U-Boot ELF
uEnv.txt ............... U-Boot environment variables
                         definitions
uImage ................. Linux kernel image for U-Boot
uramdisk.image.gz ...... Ramdisk image for U-Boot (simple
                         file system with busybox)
```

Note: only `.bashrc`, `boot.bin`, `devicetree.dtb`, `uEnv.txt`, `uImage` and `uramdisk.image.gz` are needed. The other files are provided for information and to allow to re-generate the device tree and the software components.

- Download the SD card archive from the SecBus website (`https://secbus.telecom-paristech.fr/`).
- Format a SD card with a `FAT32` first primary partition (and other partitions if you wish), make sure the partition is large enough to store the unpacked archive.
- Mount the `FAT32` partition on your PC.
- Unpack the archive in the mount point.
- Unmount the SD card.
- Configure the ZedBoard jumpers to boot from SD card (set `MIO4` and `MIO5`, unset `MIO2`, `MIO3` and `MIO6`), insert the SD card, plug the power and console USB cable and power on.

- Launch a terminal emulator like `minicom` (`minicom -D /dev/ttyACM0`), wait until Linux boots (figure 2.2) and start interacting with the bridge. Several aliases are defined for easier access to the internal registers (see the welcome banner). The running `busybox` has the `devmem` applet built in, so accessing physical addresses can also be done using `devmem`.



```
   _____           ____
  / ___/___  _____/ __ )__  _______
  \__ \/ _ \/ ___/ __  / / / / ___/
 ___/ /  __/ /__/ /_/ / /_/ (__  )
/____/\___/\___/_____/\__,_/____/
(C) Telecom ParisTech
https://secbus.telecom-paristech.fr/

-----------------------------------------------
AXI bridge commands:
  <reg-name>:       read register <reg-name>
  <reg-name> <val>: write <val> in register <reg-name>
Read-only registers:
  gpir, aw, ar, w, r, b, before_r, after_w, after_r,
  fifo
Read-write registers:
  gpor, msk, cfg, rtrig, rtrig2, rtrig3, wtrig,
  wtrig2, wtrig3, ival, iread
-----------------------------------------------
Switches value  | LEDs driven by
----------------+------------------------------
0x00            | GPOR[7:0]
0x01            | FFULL|FEMPTY|0|AWI|ARI|WI|RI|BI
0x02            | RSEQ|WSEQ|0|TEN|CEN|IEN
other           | 01010101 (0x55)
----------------+------------------------------
where AWI,ARI,WI,RI and BI are condensed 1-bit
indicators obtained by masking (bitwise boolean
AND) AW,AR,W,R and B respectively with MSK and
computing the boolean OR of the 32-bits result.
FFULL,FEMPTY,RSEQ,WSEQ,TEN,CEN,IEN are the
fields with same names in CFG.
-----------------------------------------------
#
     Offline                        NOR              VT102
```

Figure 2.2: The Linux kernel booted through the AXI bridge on a ZedBoard

Note: the CPU caches (L1 and L2) are disabled for better observability of the CPU memory accesses. Of course, this slows the CPU down; please be patient when the Linux kernel boots and loads the ramdisk image...

Note: the SD card partition from which the system booted is mounted on /mnt, so, if you added some custom files on the SD card, they are in `/mnt`.

Note: the provided bitstream embeds a Chipscope Integrated Logic Analyser (ILA) core monitoring the signals of the `M_AXI` port. It is thus possible to observe these signals from Vivado.

2.3 Experiments

First test the design in the PL by setting the switches to any configuration other than 0x00 and 0x01 (e.g. 0x02) and looking at the LEDs: if the LEDs illuminate in the 0x55 configuration things are probably OK, else the PL does not work as expected.

Reading the current status of the 8 switches:

```
zynq> gpir
0x00000002
```

Illuminating the 8 LEDs (first set the switches to 0x00):

```
zynq> gpor 0xFF
```

Configure the `MSK` register so that the LEDs blink every AXI transaction (set the switches to 0x01 so that it takes a visible effect):

```
zynq> msk 1
```

Reading the number of AXI read address requests and read responses to/from the DDR through the FPGA fabric since the beginning (returned values can be different):

```
zynq> arcnt
0x0084F9BE
zynq> rcnt
0x021AFDB8
```

Reading a 32-bits word in the DDR through the FPGA fabric (the LEDs corresponding to the AR and R transaction counters should blink):

```
zynq> devmem 0x90000000 32
0x5A51051D
```

Checking again the number of AXI read address requests and read responses:

```
zynq> arcnt
0x0085D059
zynq> rcnt
0x02312193
```

Writing a 32-bits word in the DDR through the FPGA fabric (could crash the system because we do not check first that the corresponding address in the regular address space is not used; but let us try and see what happens, the LEDs corresponding to the AW, W and B transaction counters should blink):

```
zynq> devmem 0x90000000 32 0xAAAAAAAA
```

Reading it again:

```
zynq> devmem 0x90000000 32
0xAAAAAAAA
```

Checking the number of AXI read address requests, write address requests, write data requests, read responses and write responses:

```
zynq> arcnt
0x00864F51
zynq> awcnt
0x00953812
zynq> wcnt
0x01CB8965
zynq> rcnt
0x0247D47B
zynq> bcnt
0x0086CEB8
```

Power off

```
zynq> poweroff
The system is going down NOW!
Sent SIGTERM to all processes
Sent SIGKILL to all processes
Requesting system poweroff
reboot: System halted
```

2.4 Errors, crashes and freezes

If you play a bit with the bridge an perform read and write accesses randomly with `devmem`, you will probably encounter some problems (errors, crashes, freezes and other undesirable behaviours):

- First, as explained above, accessing unmapped addresses in `CAS` or writing a read-only register raises an error.
- But you can also overwrite an important memory location, currently used by the Linux kernel. And you can do this using one or the other of the two equivalent `AAS` and `RAS`.
- Last but not least, you can also fall in an address range that is mapped in `RAS` but not in `AAS`. Indeed, due to the specificities of the Zynq architecture, `AAS` and `RAS` are not strictly equivalent. Avoiding accesses to the first MB of `AAS` (`[0x8000_0000..0x8010_0000[`) should protect you against this.

2.5 Building the whole example from scratch

If you have a SecBus distribution already installed:

```
$ cd secbus/vhdl/hsm/src/axi_bridge
$ make help
```

and follow the instructions.

Bibliography

[1] Xilinx all programmable socs: `http://www.xilinx.com/products/silicon-devices/soc.html`.

[2] Zedboard community-based web site: `http://zedboard.org/`.