

AXI bridge documentation

September 3, 2015

Contents

Introduction	ix
1 User guide	1
1.1 Interface	1
1.2 Internal registers	1
1.3 Functional description	3
1.4 Layout of the internal registers	5
2 Application notes	7
2.1 Introduction	7
2.2 Configuring the ZedBoard with the bridge	8
2.3 Experiments	11
2.4 Errors, crashes and freezes	13
2.5 Building the whole example from scratch	13

List of Tables

1	Revision history	ix
1.1	AXI bridge input and output ports	1
1.2	AXI bridge generic parameter	2
1.3	AXI Bridge registers table	2
1.4	Value of GPO as a function of GPIR	4

List of Figures

1.1	AXI bridge	4
1.2	AXI Bridge <code>gpi_r</code> register layout: General Purpose Input Register . .	5
1.3	AXI Bridge <code>gpo_r</code> register layout: General Purpose Output Register .	5
1.4	AXI Bridge <code>cfg</code> register layout: ConFiGuration	5
2.1	AXI bridge in ZedBoard	8
2.2	The Linux kernel booted on a ZedBoard	11

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 bridge from a pure functional perspective will find the description of the internal registers and their role.

1.1 Interface

The 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 synchronous active high reset (`SRST`). Table 1.1 lists the input and output ports.

Name	Direction	Bit-width		Description
		Data	Address	
<code>ACLK</code>	input	1	-	System clock
<code>ARESETN</code>	input	1	-	System reset
<code>SRST</code>	input	1	-	Registers synchronous reset
<code>S0_AXI</code>	-	32	12	AXI3 lite slave interface
<code>S1_AXI</code>	-	32	32	AXI3 slave interface
<code>M_AXI</code>	-	32	32	AXI3 master interface
<code>GPI</code>	input	8	-	General Purpose Input
<code>GPO</code>	output	8	-	General Purpose Output

Table 1.1: AXI bridge input and output ports

The bridge can be customized with 1 generic parameter (table 1.2).

1.2 Internal registers

The bridge embeds a set of 32-bits internal registers (table 1.3). Their default value is defined in the `bitfield_pkg.vhd` package. All registers are initialized to their

Name	Type	Default
<code>S0_AXI_ADDRESS_WIDTH</code>	Address width of S0_AXI port	12

Table 1.2: AXI bridge generic parameter

default value when the `SRST` synchronous reset is asserted high or when the system reset is asserted low. 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
<code>gpir</code>	0x0	r	Current value of GPI input (8 LSBs only)
<code>gpor</code>	0x4	rw	Value sent on General Purpose Output when GPI=0x01 (8 LSBs only)
<code>msk</code>	0x8	rw	Used to compute the one-bit activity indicators from the AXI transaction counters
<code>aw</code>	0xc	r	Counts the completed transactions on the Address Write AXI channel
<code>ar</code>	0x10	r	Counts the completed transactions on the Address Read AXI channel
<code>w</code>	0x14	r	Counts the completed transactions on the Write data AXI channel
<code>r</code>	0x18	r	Counts the completed transactions on the Read data AXI channel
<code>b</code>	0x1c	r	Counts the completed transactions on the Write response AXI channel
<code>cfg</code>	0x20	rw	Configuration and status register
<code>before_r</code>	0x24	r	Last rdata value before trigger
<code>after_w</code>	0x28	r	First wdata value after trigger
<code>after_r</code>	0x2c	r	First rdata value after trigger
<code>rtrig</code>	0x30	rw	First read data value of trigger pattern
<code>rtrig2</code>	0x34	rw	Second read data value of trigger pattern
<code>rtrig3</code>	0x38	rw	Third read data value of trigger pattern
<code>wtrig</code>	0x3c	rw	First write data value of trigger pattern
<code>wtrig2</code>	0x40	rw	Second write data value of trigger pattern
<code>wtrig3</code>	0x44	rw	Third write data value of trigger pattern
<code>ival</code>	0x48	rw	Value to inject

Name	Address	Dir.	Description
iread	0x4c	rw	The read value to overwrite during injection
fifo	0x50	r	FIFO read register
offset	0x54	rw	OFFSET to add to S1_AXI addresses during S1_AXI -> M_AXI translation

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 value of the OFFSET register is added to the 32 bits addresses:

```
M_AXI.AWADDR <= S0_AXI.AWADDR + OFFSET;
M_AXI.ARADDR <= S0_AXI.ARADDR + OFFSET;
```

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, if the CFG.CNSTRG (CoNSecutive TRiGger) flag is set, the corresponding sub-trigger is reset and the matching process restarts from the beginning. Else, if the CFG.CNSTRG flag is clear (the default), the matching process continues until the specified number of matches is reached. 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, RTRIG_x, WTRIG_x) 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 downto 0)	GPO source
0x00	GPOR (7 downto 0)
0x01	FFULL FEMPTY 0 AWI ARI WI RI BI
0x02	RSEQ WSEQ CNSTRG TEN CEN IEN
other	"01010101" (0x55)

Table 1.4: Value of GPO as a function of GPIR

Figure 1.1 represents the bridge.

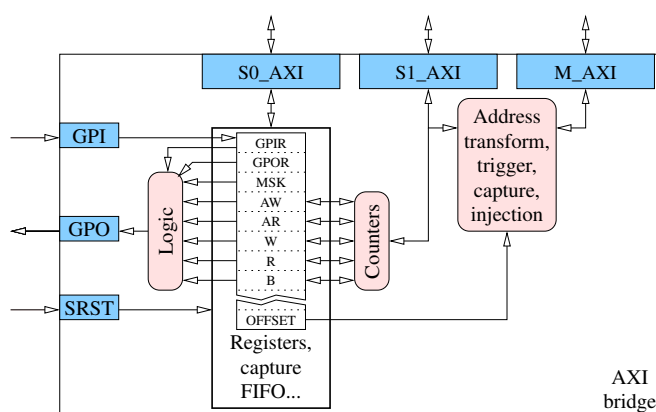


Figure 1.1: AXI bridge

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 1.2: AXI Bridge gpi_r register layout: General Purpose Input Register



Figure 1.3: AXI Bridge gpo_r 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 bridge is integrated in the Programmable Logic (PL) of the Zynq core. The goal is to use the bridge to create 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. Moreover, 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 `S0_AXI` slave port of the bridge is connected to the `M_AXI_GP0` master port of the PS. The CPU accesses the bridge internal registers in the `[0x4000_0000, 0x8000_0000[` address range, denoted Control Address Space (CAS) in the following. Of course the CAS is not fully mapped: it is limited to the small number of existing registers.

The `S1_AXI` slave port of the bridge is connected to the `M_AXI_GP1` master port of the PS and the `M_AXI` master port of the bridge is connected to the `S_AXI_HP0` slave port of the PS. The address transform applied by the bridge is defined by the `OFFSET` internal register, which default value is set to `0x8010_0000` for this application note. Warning: `OFFSET` is a read-write register and its value can be changed. This possibility **must not** be used for this application note. The `OFFSET` value **must** remain the default value from boot to power off. This offset maps the `[0x8000_0000, 0x9ff0_0000[` address range, denoted Alternate Address Space (AAS) in the following, to the `[0x0010_0000, 0x2000_0000[` address range, that is, the 511 high MBs of DDR, denoted Regular Address Space (RAS) in the following. The 1 MB shift is intentional: the memory map of the Zynq system is such that the PL cannot access the first MB. Each memory location of the DDR has two addresses: one in the RAS and the other in the AAS, except for the first MB of DDR (`[0x0000_0000, 0x0010_0000[`) which is accessible only in the RAS. The main goal of this application note is to show how a complete GNU/Linux software stack can boot and run using only the AAS.

The `GP_I` primary input is connected to the 8 switches of the ZedBoard and the `GP_O` primary output is connected to the 8 user LEDs. So, the `ARI`, `AWI`, `WI`, `RI` and `BI`

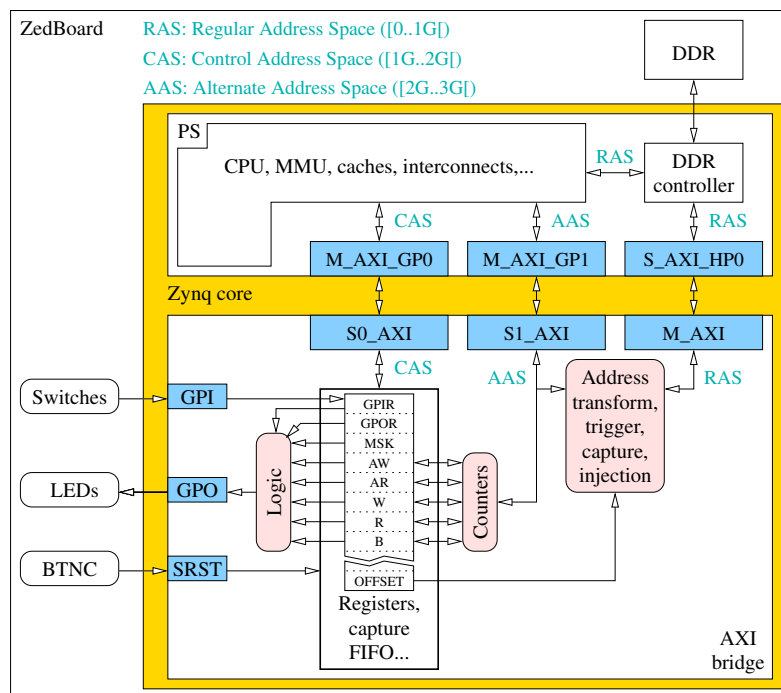


Figure 2.1: AXI bridge in ZedBoard

AXI activity indicators drive the LEDs in the `GPIR=0x01` mode and it is possible to observe the `S1_AXI` to `M_AXI` activity. Writing 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.

The `SRST` synchronous reset is connected to the centre (BTNC) button of the 5 press buttons pad. Pressing the centre button of the 5 buttons pad of the ZedBoard resets the internal registers to their default value for this application note, as defined in the `bitfield_pkg.vhd`: 0, except the `OFFSET` register, which default value is `0x8010_0000`.

2.2 Configuring the ZedBoard with the bridge

The provided SD card archive contains all files needed to run a `busybox` on top of a Linux kernel. It comprises several Linux kernels and device tree blobs. Depending on the boot options, it is possible to run the software stack either in the RAS or the AAS and with or without the L1/L2 CPU caches enabled (4 possibilities). The SD card archive also contains two initramfs root file systems: one with a GDB server, all binaries unstripped, compiled in debug mode, without optimizations, and the other, smaller, with optimized and stripped binaries. When choosing to boot in RAS mode, all memory accesses are sent directly to the DDR controller. The device tree blob and the Linux kernel image are customized such that the available DDR memory seen by the Linux kernel is entirely in the RAS, in the `[0x0000_0000, 0x2000_0000[`

range. When choosing the AAS mode, all memory accesses are routed to the AXI simple bridge in the PL. The device tree blob and the Linux kernel image are customized such that the available DDR memory seen by the Linux kernel is entirely in the AAS and limited to the `[0x8000_0000, 0x9ff0_0000[` range. The last MB (`[0x9ff0_0000, 0xa000_0000[`) is excluded for the reasons exposed in the previous section.

The SD card archive content is:

```

axi_bridge.pdf ... Documentation
.bashrc ..... Initialization script
                    (aliases definitions)
boot.bin ..... Zynq boot image (FSBL,
                    bitstream and U-Boot)
COPYING ..... License (English)
COPYING-FR ..... Licence (French)
devicetree-aas.dtb ..... Device tree blob
devicetree-aas-nocache.dtb ..... Device tree blob
devicetree-ras.dtb ..... Device tree blob
devicetree-ras-nocache.dtb ..... Device tree blob
initramfs-debug.rootfs ..... Initramfs
initramfs.rootfs ..... Initramfs
kernel-aas-nocache.uImage ..... Linux kernel image
kernel-aas.uImage ..... Linux kernel image
kernel-ras-nocache.uImage ..... Linux kernel image
kernel-ras.uImage ..... Linux kernel image
README ..... Short documentation
src/ ..... Source files for reference
    bitstream.bit ..... Bitstream
    devicetree-aas_dts/ ..... Device tree sources
        device-tree.mss
        pl.dtsi
        skeleton.dtsi
        system.dts
        zynq-7000.dtsi
    devicetree-aas-nocache_dts/ . Device tree sources
        device-tree.mss
        pl.dtsi
        skeleton.dtsi
        system.dts
        zynq-7000.dtsi
    devicetree-ras_dts/ ..... Device tree sources
        device-tree.mss
        pl.dtsi
        skeleton.dtsi
        system.dts
        zynq-7000.dtsi
    devicetree-ras-nocache_dts/ . Device tree sources
        device-tree.mss
        pl.dtsi
        skeleton.dtsi

```

```

        system.dts
        zynq-7000.dtsi
    fsbl.elf ..... First Stage Boot Loader ELF
    top.hdf ..... Hardware Description File
    u-boot.elf ..... U-Boot ELF
    uEnv.txt ..... U-Boot environment variables

```

Note: only `.bashrc`, `boot.bin`, the device tree blobs, the kernel images, the `initramfs` images and `uEnv.txt` are needed. The other files are provided for information and to allow to re-generate the device tree and the software components.

- Prepare a SD card from which the ZedBoard will boot. Create a FAT32 first partition and make it large enough for the provided archive (you can create more partitions if you wish). Mount it on your host PC. In the following we assume its mount point is `/media/SDCard`.
- Download the `sdcard-axi-bridge.tgz` SD card archive from the SecBus website (<https://secbus.telecom-paristech.fr/>).
- Unpack the archive in the SD card:

```
tar --directory=/media/SDCard -xf sdcard-axi-bridge.tgz
```

- Unmount the SD card, plug it to the ZedBoard, configure the jumpers to boot from the SD card and connect the USB-UART cable to your host PC.
- Power on the ZedBoard and launch a serial console on your host PC (minicom, `cu`, `putty`...):

```
minicom -D /dev/ttyACM0
```

- Stop the U-Boot countdown, load the environment variables from the SD card, list them and (optionally) save them on the QSPI flash:

```
secbus-uboot> run uenvboot
secbus-uboot> printenv
secbus-uboot> saveenv

```

- Eight environment variables are defined to boot in eight different configurations (AAS or RAS, with / without caches, regular / debug `initramfs`). Select one and run it:

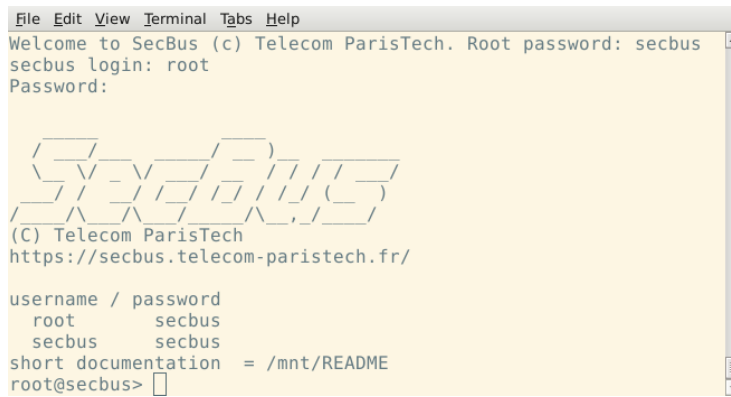
```

secbus-uboot> run ras
secbus-uboot> run aas
secbus-uboot> run ras-nocache
secbus-uboot> run aas-nocache
secbus-uboot> run ras-debug
secbus-uboot> run aas-debug
secbus-uboot> run ras-nocache-debug
secbus-uboot> run aas-nocache-debug

```

- Wait until Linux boots. Note: if you selected a boot mode with the L1/L2 CPU caches disabled the boot sequence takes significantly longer. Please be patient.

Log in either as `root` or the regular `secbus` user (password `secbus` in both cases). You are done (figure 2.2) and you are running a minimal GNU/Linux OS. If you chose an AAS boot mode, all accesses to the external memory are routed to the PL. You can now start interacting with the bridge and observe its activity thanks to its internal registers. Several aliases are defined to ease the interactions with the bridge, as summarized in the `/mnt/README` file.



```

File Edit View Terminal Tabs Help
Welcome to SecBus (c) Telecom ParisTech. Root password: secbus
secbus login: root
Password:

          _ _ _ _ _
         / / / / /
        / / / / /
       / / / / /
      / / / / /
     / / / / /
    / / / / /
   / / / / /
  / / / / /
 / / / / /
/ / / / /

(C) Telecom ParisTech
https://secbus.telecom-paristech.fr/

username / password
  root      secbus
  secbus    secbus
short documentation = /mnt/README
root@secbus>

```

Figure 2.2: The Linux kernel booted on a ZedBoard

Notes:

- The running `busybox` has the `devmem` applet built in, so accessing physical addresses can be done using `devmem`. The `secbus` regular user has `devmem` privilege.
- 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`.
- The bitstream embeds a Chipscope Integrated Logic Analyzer core allowing to observe the `M_AXI` signals from Vivado.

2.3 Experiments

In this section we assume that the ZedBoard was booted in AAS mode:

```
secbus-uboot> run aas
```

such that all memory accesses are routed to the AXI simple bridge.

Let us first test the design in the PL by setting the switches to any configuration other than `0x00`, `0x01` or `0x02` (e.g. `0x03`) 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:

```
root@secbus> gpir
0x00000002
```

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

```
root@secbus> 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):

```
root@secbus> 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):

```
root@secbus> ar
0x0084F9BE
root@secbus> r
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):

```
root@secbus> devmem 0x90000000 32
0x5A51051D
```

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

```
root@secbus> ar
0x0085D059
root@secbus> r
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):

```
root@secbus> devmem 0x90000000 32 0xAAAAAAAA
```

Reading it again:

```
root@secbus> devmem 0x90000000 32
0xAAAAAAAA
```

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

```
root@secbus> ar
0x00864F51
root@secbus> aw
0x00953812
root@secbus> w
0x01CB8965
root@secbus> r
0x0247D47B
root@secbus> b
0x0086CEB8
```

Power off

```
root@secbus> 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 and 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.

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/>.