XAPP515 (v1.0) May 19, 2006

Using Xilinx m4 Functions to Write Bus Functional Language Stimuli for CoreConnect Buses

Author: Lester Sanders

Summary

This application note provides definitions and examples of Xilinx developed m4 functions used to create Bus Functional Language (BFL) files. BFL files are used in the Bus Functional Model (BFM) simulation of Xilinx Processor Intellectual Processor (PIP) cores. These BFMs allow the simulation of PIP cores which interface to the IBM CoreConnect™ Processor Local Bus (PLB) and On-chip Peripheral Bus (OPB).

Introduction

The m4 (GNU m4) preprocessor is used as a front end for creating BFL test code. These BFL files are stimulus files for the Bus Functional Models located in the `<EDK>/hw/XilinxBFMinterface/pcores` directory. The examples in `xapp516.zip` include BFL and VHDL test code to provide stimuli for BFM simulation of PIP cores. The `<core>.m4` and `<core>_defs.m4` files make extensive use of the m4 macros listed in this note. The macros are defined in `<EDK>/gnu/m4/proc_defs_opb.m4` (`proc_defs_plb.m4`).

Core Definitions File

The first step in writing BFL is to write the `<core>_defs.m4` definition file. The `<core>_defs.m4` file defines the generics and register locations of the Processor IP under test.

Defining Generics

The values of the generics in `<core>_defs.m4` must be the same as the values provided in the `<core>_tb.vhd` file. In a BFM simulation in the EDK environment, the `<core>_tb.vhd` entity is instantiated by `/simulation/behavioral/my_core_wrapper.vhd`, in which C_BASEADDR is `0x30000000`. An example of the definition of generics for the OPB GPIO core is shown in "Defining Generics inopb_gpio_defs.m4."

Defining Generics inopb_gpio_defs.m4

```
define('opb_gpio_generic_defs', '
ifdef('C_OPB_DWIDTH',,'define('C_OPB_DWIDTH','32')')
ifdef('C_OPB_AWIDTH',,'define('C_OPB_AWIDTH','32')')
ifdef('C_BASEADDR',,'define('C_BASEADDR','0x30000000')')
ifdef('C_HIGHADDR',,'define('C_HIGHADDR','0x300000FF')')
ifdef('C_GPIO_WIDTH',,'define('C_GPIO_WIDTH','32')')
ifdef('C_INTERRUPT_PRESENT',,'define('C_INTERRUPT_PRESENT', '1')')
ifdef('C_ALL_INPUTS',,'define('C_ALL_INPUTS','0')')
ifdef('C_IS_BIDIR',,'define('C_IS_BIDIR', '1')')
ifdef('C_DOUT_DEFAULT',,'define('C_DOUT_DEFAULT', '0x00000000')')
ifdef('C_TRI_DEFAULT',,'define('C_TRI_DEFAULT', '0xFFFFFFFF')')
ifdef('C_IS_DUAL',,'define('C_IS_DUAL','1')')
```

Defining Register Locations

The register address locations are usually defined relative to the C_BASEADDR generic, with values given in the data sheet for the PIP core. The **add_offset** macro adds two hex numbers to form an address. Unlike the m4 evaluation function, which returns a signed 32-bit result, the add_offset macro creates unsigned 32-bit results (Table 1).

Table 1: **add_offset Macro Examples**

m4	BFL
add_offset(C_BASEADDR,0x1C)	0x3000001C
add_offset(0xFFFFFFF1,0xF)	0x00000000

There are several register definition macros. The **set_reg_default** macro is similar to the BFL set_alias command.The **def_reg** macro defines a register mnemonic and a default value for the register. The default value is 0x00000000. This macro checks to see if an address has been provided (Table 2).

Table 2: **Register Macro Examples**

m4	BFL
set_reg_default('RBR',0xABCD)	RBR_DEFAULT_VALUE=0xABCD
def_reg(DATA)	def_reg(DATA,0x00000000)
DATA DATA_DEFAULT_VALUE	0x00000000 0x00000000
def_reg(DATA, 0x3000000, make_num)	0x30000000 0x3B51FE03

Registers are given symbolic names and addresses for the OPB GPIO in the opb_gpio_defs.m4 file:

```
define('gpio_registers',
'((DATA, add_offset(C_BASEADDR, 0x00), 0x00000000),
  (TRI,  add_offset(C_BASEADDR, 0x04), C_TRI_DEFAULT),
  (DATA2, add_offset(C_BASEADDR, 0x08), 0x00000000),
  (TRI2,  add_offset(C_BASEADDR, 0x0C), C_TRI_DEFAULT_2),
  (GIE_REG, add_offset(C_BASEADDR, 0x11C), 0x00000000),
  (ISR_REG, add_offset(C_BASEADDR, 0x120),0x00000000),
  (IER_REG, add_offset(C_BASEADDR, 0x128),0x00000000))')
```

Core Stimuli File

The core stimuli file has the following sections.

- Setup
- Check reset value of registers
- PLB/OPB Write
  - Generate stimuli to registers.
  - Request VHDL test bench to verify PIP core results.
- PLB/OPB Read
  - Request VHDL test bench to generate external stimuli.
  - Verify PIP core results

The setup section of opb_gpio.m4 is:

```
include('proc_defs_opb.m4')
include('opb_gpio_defs.m4')
divert(-1)
opb_gpio_generic_defs
make_registers('gpio_registers')
set_dev
start
```

The include statements allow `proc_defs_opb.m4` and `<core>_defs.m4` to be included in `opb_gpio.m4`. The `proc_defs_opb.m4` is a file in the EDK install tree which contains predefined macros which make the m4 easier to read and write. The core generics and registers are defined in the `<core>_defs.m4` file.

The **make_registers** macro creates a set of registers from a list of registers, with optional default values. An example of the make_registers macro in `opb_gpio.m4` is given in the setup example above. The set_dev macros are defined in Table 3. The set_device construct in BFL defines the device to which the commands in the BFL apply.

Table 3: **set_dev Macro Examples**

m4	BFL
set_dev	set_device(path=/bfm_system/bfm_memory/bfm_memory, device_type=opb_device)
set_dev(device1)	set_device(path=/bfm_system/bfm_memory/bfm_memory, device_type=opb_device)
set_dev(monitor)	set_device(path=/bfm_system/bfm_memory/bfm_memory,opb_mon, device_type = opb_monitor)

Reading/Writing Address Locations

To read and write registers, the read_word/write_word macros are used. In addition, examples are provided. The **read_word** macro creates a BFL read command. Byte enables are calculated and checked. The **read_word** macro accepts an optional third argument which is passed through unchanged. As an example of the third argument, BFL command qualifiers such as req_size and req_delay can be included in the m4 file. The **read_word** macro requires a data comparison value. To perform a BFL read with no data comparison, use raw BFL. The **read_half** and **read_byte** macros behave identically to **read_word** for half word and byte reads.

The **write_word** macro creates a BFL write command. Byte enables are calculated and checked. The **write_word** macro accepts an optional third argument which is passed through unchanged. The **write_word** macro requires a data comparison value. To perform a BFL write with no data comparison, use raw BFL. The **write_half** and **write_byte** macros are similar to **write_word** for half word and byte writes.

Table 4 provides m4/BFL examples of **read_word** and **write_word** commands. The r**read_word** and **write_word** macros are often a function of the generics. If the GPIO is configured as a single channel device, there is no need to read the DATA2 register:

```
ifelse(C_IS_DUAL, 1,'
  read_word(DATA2, mask_reg(DATA2_DEFAULT_VALUE, C_GPIO_WIDTH))
```

Table 4: **read_word/write_word Examples**

m4	BFL
read_word(IIR, IIR_DEFAULT_VALUE)	read(addr=30001008, be=11110000, data=00001010)
read_word(IIR, IIR_DEFAULT_VALUE, 'req_size=4,req_delay=5')	read(addr=30001008, be=11110000, data=000000001, req_size=4, req_delay=5)
read(IIR,calc_be(IIR, word))	read(0x30001008, be=11110000)
write_word(IIR,IIR_DEFAULT_VALUE,'req_size=4,req_delay=5')	write(addr=30001008, be=11110000, data=00000001, req_size=4,req_delay=5)
write_word(IIR,calc_be(IIR,word))	write(0x30001008, be=11110000)

Send/Wait Macros

The send/wait macros provide a method to synchronize BFL with VHDL test benches. The in/out direction is relative to the VHDL test bench. The *_in macros send a synchronization pulse to the VHDL test bench, the *_out macros wait for a signal from the VHDL test bench. All send commands also reset the test bench watchdog timer.

Table 5: **Send/Wait Macros**

m4	BFL
send/wait	send/wait
assert	send(level=5, level=9) wait(level=6)
assert_in	send(level=5, level=9)
assert_out	wait(level=6)
assign	send(level=7, level=9) wait(level=8)
assign_in	send(level=7, level=9)
assign_out	wait(level=8)
nop	send(level=0, level=9)
reset_wdt	send(level=9)
nop	send(level=0, level=9)
reset_wdt	send(level=9)
start	wait(level=1)
stop	send(level=2, level=9)
wait	send(level=3,level=9) wait(level=4)
wait_for_intr	wait(level=31)

The values of level in the BFL column correspond to the signals on the SYNCH_OUT[0:31] and SYNCH_IN[0:31] buses. The start macro causes a pulse on SYNCH_OUT(1), and a wait_for_intr causes a pulse on SYNCH_OUT(31).

The BFL cannot efficiently write status messages to the simulator transcript window such as “Checking initialization values” to the simulator transcript window. To do this, the BFL assert macro causes control to transfer to the VHDL test bench, which writes the status message.

Similarly, when the BFL needs the VHDL to provide stimuli, the assign_in macro is used.

An example of the interaction between the stimuli generated by m4 and the results tested by the VHDL is shown in “m4 - VHDL Testing.” In this example, the BFM writes all Fs to the GPIO data register, followed by all 0s.The VHDL writes a status message to the transcript window

indicating the test being done. The VHDL Then verifies that the all Fs followed by all 0s patterns are written out GPIO_IO_O.

m4 - VHDL Testing

```
m4:

write_word(DATA, GPIO_ALL_ONES)
  assert_in
  write_word(DATA, ALL_ZEROS)
  assert_in
VHDL
assert FALSE report " Checking that data register writes 1s severity NOTE;
for i in 0 to C_GPIO_WIDTH-1 loop
assert GPIO_IO_O(i) = '1' report "GPIO_IO_O did not write 1s correctly"
severity ERROR
end if;
end loop;
assert_in(1,opb_clk,synch_in(5 to 5));
assert FALSE report "Checking that data register writes 0s" severity NOTE;
```

An example of the wait_for_intr macro from the opb_uartlite_bfm design uses repeated sections of the following code.

```
 wait_for_intr
read_word(STATUS,  0x00000011)
read_word(RXFIFO, adjust_for_numbits(0x00000004))
write_word(TXFIFO, 0x00000006)
read_word(STATUS, 0x00000010)
```

The interrupt from the opb_uartlite drives the sig_dev_intr signal in opb_uartlite_tb.vhd. In this test bench, the transmitter loops back to the receiver. The test cycles through the transmission/reception of data, with the interrupt prompting the BFL to check/generate stimuli.

An example of using a loop operation to fill the transmitter memory is given below:

```
forloop('i', 0, 507,
'write_word(add_offset(C_BASEADDR, (i*4)),0xFFFFFFFF)
)
```

A second example of writing and then reading memory in a loop is given below. This uses the built in m4 **eval** macro to calculate two hex digits to create the 32-bit value. The 0x"eval(i*4, 16,2) produces 2 hex digits in hex (16) of the i*4 value:

```
forloop('i', 0, 10,
'write_word(add_offset(C_BASEADDR, (i*4)),0x"eval(i*4, 16,
2)"eval((i*4)+1, 16
, 2)"t6eval((i*4)+2, 16, 2)"eval((i*4)+3, 16, 2))'
)
```

In the above loop, the first iteration of 0x"eval(i*4, 16, 2)"eval((i*4)+1, 16, 2)"eval((i*4)+2, 16, 2)"eval((i*4)+3, 16, 2) argument is produces a value of 0x00010203.

Miscellaneous Macros

This section contains the definition and examples of miscellaneous macros.

The m4 macro processor provides the means to perform arithmetic in any base, up to and including base 36. These macros, and some conventions associated with them are described below. All addresses and data values are hex numbers of the form:

0[xX][0-9a-zA-Z]* with up to 8 hex digits.

For example:

```
0xA 0Xa 0xabcdef12 0XabcDEF
```

The **make_num** macro produces a different 32-bit value on each invocation, as shown in Table 6.

Table 6: **make_num Examples**

m4	BFL
make_num	0x7D53041F
make_num	0x290D33B3

The **strip_hex** macro removes the 0x or 0X prefix from a hex number. This is used to format hex numbers for BFL or prior to concatenating hex strings together. Examples are shown in Table 7.

Table 7: **strip_hex Examples**

m4	BFL
strip_hex(0xABCD)	ABCD
strip_hex(make_num)	A53B6439
strip_hex(make_num)'strib_hex(make_num)	72F586351E1BF095

The **make_hex** macro converts a number in another base into a hex number, as shown in Table 8.

Table 8: **make_hex Examples**

m4	BFL
make_hex(1)	0x00000001
make_hex(249)	0x000000F9
make_hex(0b1010)	0x0000000A
make_hex(0r36:z9)	0x000004F5

The **last_digit** macro returns the last digit of a number. It is used to check the validity of an address, as shown in Table 9.

Table 9: **last_digit Examples**

m4	BFL
last_digit(0xA)	A
last_digit(0xabcdef12)	2
last_digit(make_num)	1

The **calc_be** macro accepts 2 arguments, an address and a transaction type, and returns a byte-enable value. If an illegal address for the given transaction type is found, calc_be prints an error message to the screen, as shown in Table 10.

Table 10: **calc_be Examples**

m4	BFL (be)
calc_be(0x00000000, word)	11110000
calc_be(0x00000004, word)	00001111
calc_be(0x00000003, word)	no word be
calc_be(0x00000008, half)	11000000
calc_be(0x00000005, half)	no half word be

Table 10: **calc_be Examples**

m4	BFL (be)
calc_be(0x00000000,byte)	10000000
calc_be(0x0000000F, byte)	00000001

Predefined constants are given in Table 11.

Table 11: **Predefined Constants**

m4	BFL (be)
ALL_ZEROS	0x00000000
ALL_ONES_BYTE_0	0xFF000000
ALL_ONES_BYTE_1	0x00FF0000
ALL_ONES_BYTE_2	0x0000FF00
ALL_ONES_BYTE_3	0x000000FF
ALL_ONES_HALF_0	0xFFFF0000
ALL_ONES_HALF_1	0x0000FFFF
ALL_ONES_WORD	0xFFFFFFFF

References

GNU m4, version 1.4—Rene Seinda

XAPP516, *Bus Functional Model Simulation of Processor Intellectual Property*

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
05/08/06	1.0	Initial Xilinx release.