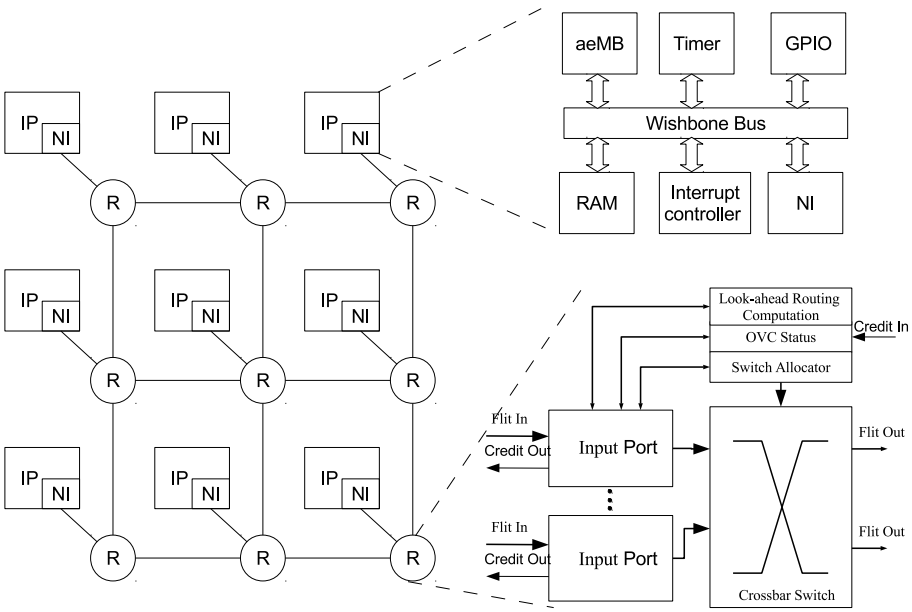
NoC based MPSoC User manual

Alireza Monemi
alirezamonemi@opencores.org



Feb 2014

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Network on chip (NoC) has been proposed [1] to eliminate the problem of traditional bus based intercommunication in dealing with large number of IPs. This project aims to propose an open source NoC based multiprocessor system on chip (MPSoC). The processing cores are connected via a low latency NoC network router. The NoC router is a two clock-cycles pipelined wormhole virtual channel router. The router first pipeline stage is the parallel look-ahead route computation with a non-speculative VC/switch allocation. The second stage is the crossbar switch traversal.

A processing tile consist of an aeMB [2] processor, the programming RAM and other optional peripheral devices such as timer, GPIO, external interrupt and interrupt controller. All of mentioned components are connected via wishbone bus inside the processing tile. The MPSoC can be generated with processing tile with different type and number of peripheral devices, simply by defining of parameters in the top level entity file.

All the projects RTL codes are written using synthesizable Verilog HDL language. We test our design on Altera DE2-115 educational board [3]. The code is compiled on a PC running Linux using Quartus II version 13.0.0 and verified using Modelsim Altrea version.

You may find this project helpful in the following three main ways:

1.1 NoC based MPSoC

Selecting the src/MPSoC_top.v as the top level entity will implement the NoC based MPSoC. The design parameters such as number of cores, NoC topology and routing algorithm, number of VC and number of peripheral devices for each individual core can be defined in src/define.v file. The proposed MPSoC can be used to test a real life application such as MPEG decoder.

The software code for each individual processor must be written in sw/mpsoc_code/cpuX_Y.c where X and Y are the location of the core in x-axis and y-axis respectively. Running the sw/compile_mpsoc will compile all the processor codes and store the generated mif files in sw/ram folder. It also copies the generated mif files in simulation folder which you can simulate the design using src/testbench_mpsoc.v file. If you are implementing the design on an FPGA device, u can program all processors using sw/prog_memories.

1.2 Single processor wishbone based SoC:

You can make a wishbone based SoC by selecting the src/SoC_IP_top.v as top level entity. This file implement just one processing tile with no NoC network interface. The peripheral devices can be easily configured by defining parameters. The software code must be located in sw/soc_code/soc.c and can be compiled using sw/compile_soc.

1.3 Low latency NoC simulator

In case that you just want to simulate NoC infrastructure in dealing with different type of traffic e.g. random or hot-spot without running the software on aeMB, You can set the following parameter NI_CTRL_SIMULATION = "testbench" in Simulation file. Setting this parameter as ”testbench” will remove the aeMB processors in simulation file and will leave handling the NI control signals to the testbench file. The tasks.v file contains two tasks one for sinking and another for injecting NoC packets to the network. A simple testbench file has been written in src/testbench_noc.v file.

1.4 Configuring the processing tiles

Each tile can be configured with different type or number of peripheral devices. You can configure each core inside the MPSoC with different type of available peripheral devices. You can also easily add new peripheral devices to the processing cores.

1.4.1 Enabling the available peripheral device

Several peripheral devices such as timer, gpio, interrupt controller is included in the design project which can be enabled simply by defining the peripheral device enable parameter at top level entity. Different parameters can be passed to each processing tile by defining the peripheral device parameter as follow:

```
parameter PERIPHERAL_ARRAY = "IPx1_y1:value1; IPx2_y2:
value2;Def:[default value for other cores]"
```

where IPm_n is the IP core located in x=m and y=n. e.g:

```
parameter TIMER_EN_ARRAY = "IP0_0:1;IP0_1:1;Def:0"
```

will enable the timer for processing cores located in 0,0 and 0,1 location while the rest of cores will be configured with no timer enabled.

All values will be passed to the submodule as string. By using following function:

```
ip_value(x,y,PERIPHERAL_ARRAY)
```

where the x and y is the address of the core and PERIPHERAL_ARRAY is the top level parameter holding the value for all cores.

for GPIO you can define the number of port and port_width just by defining the GPIO_WIDTH parameter. e.g:

```
O_PORT_WIDTH_ARRAY = "IP0_0:7,7,7,7,7,7,7,7;Def:1"
```

will define 8 ports with the size of 7 bit each for core (0,0) and all other cores will have just one output port width the size of 1 bit. Note that the maximum port width is 32.

All wishbone bus parameters will be set automatically by the program.

1.4.2 Adding new peripheral device to the design

If you aim to add a new peripheral device to the processing tile while intended to make it as flexible of predefined peripheral devices, you need to do following steps:

1. copy your wishbone adaptable peripheral device(s) Verilog file in `/src/IP_CORE`
2. Include your new device in the `/src/IP_CORE_aeMB_IP.v` file:

```
if (NEW_DEV_EN) begin :new_dev_gen
  new_device #(
   .ADDR_WIDTH(NEW_DEV_ADDR_WIDTH)
  )
  the_new_dev
  (
    .clk   (clk),
    .reset (reset),

    //slave_port
    .s_dat_i  (slave_dat_i  [NEW_DEV_ID]),
    .s_sel_i  (slave_sel_i  [NEW_DEV_ID]),
    .s_addr_i (slave_addr_i [NEW_DEV_ID),
    .s_stb_i  (slave_stb_i  [NEW_DEV_ID]),
    .s_we_i   (slave_we_i   [NEW_DEV_ID]),
    .s_cti_i  (slave_tag_i  [NEW_DEV_ID]),
    .sa_dat_o (slave_dat_o  [NEW_DEV_ID]),
```

```
.sa_ack_o (slave_ack_o  [NEW_DEV_ID]),
//master port (if any)
.m_dat_i  (master_dat_i [NEW_DEV_ID]),
.
.
//Other ports
.
.
.
}
```

3. Update the `/src/parameter.v` file :
update the number of master and slave port on wishbone bus as follow:

```
MASTER_NUM = [old MASTER_NUM] + (NEW_DEV_EN *
    number of master port the new device has)

SLAVE_NUM  = [old SLAVE_NUM]  + (NEW_DEV_EN *
    number of slave port the new device has)
```

`NEW_DEV_EN` is a parameter which is passed to each core and indicate if the desired peripheral device is enabled for the core or not.

Define the address range for your device

```
localparam NEW_DEV_START = LAST_DEV_START +
                           LAST_DEV_BK_NUM;
localparam NEW_DEV_BK_NUM = [number of block
                          addresses you need]
```

Each block address is 2^{24} addresses.

Finally update the device_id number:

```
localparam NEW_DEVICE_ID = (NEW_DEVICE_EN) ?
                    LAST_DEVICE_ID_E +1 : 255;
localparam NEW_DEVICE_ID_E = LAST_DEVICE_ID_E +
                    NEW_DEVICE_EN;
```

4. Update the wishbone address comparator file:
`/src/IP_core/bus_addr_cmp.v`

Add the new enable parameter at the beginning of the file:

```
parameter NEW_DEV_EN  = 1,
```

Add the new device base address at the end of the file:

```
end else if(k == NEW_DEV_ID) begin
    assign   base_start_addr [k] = NEW_DEV_ADDR_START;
    assign   base_end_addr   [k] = NEW_DEV_ADDR_START+
                                       NEW_DEV_BK_NUM;
end
```

1.5 Installing aeMB compiler

AeMB [2] is binary adaptable with Xilinx MicroBlaze processor that can be programed using `mb-gcc` compiler. For installing mb-gcc do the following steps:

1. Download the mb-gcc compiler from the following address. Unzip the file and copy it in `/opt` folder.
   http://gnuradio.org/tools/mb-gcc-4.1.1.gr2.i386.tar.gz
2. Add the mb-gcc compiler to the PATH variable. Open the `~/.bashrc` file using
   `gedit ~/.bashrc`
   and add the following line at the end of the file then save it.
   `export PATH=$PATH:/opt/microblaze/bin`
3. Install srecord
   `sudo apt-get install srecord`
4. Compile the i2hex converter
   `cd sw/compile/ihex`
   `make`

1.6 NoC

We developed a two stages, wormhole virtual channel NoC router. In NoC IP cores communicate by sending network packets. An NoC packet consists of several fixed size flow control digit (flits). There are three type of flits : header flit, buddy flit and tail flit. The minimum packet size is 2 flits,(one header and one tail).

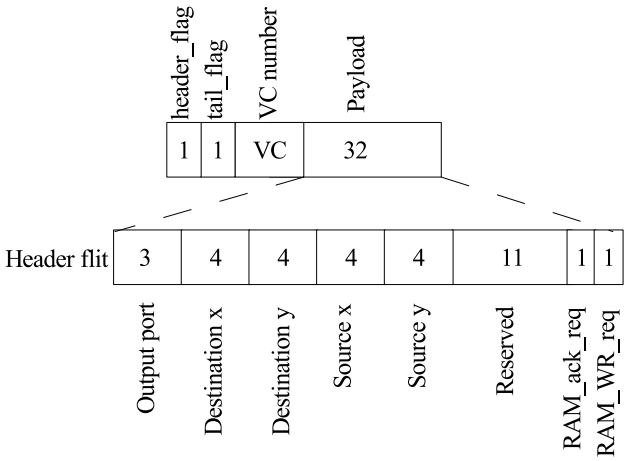


Figure 1.1: The NoC packet format

Figure 1.1 illustrates the format of NoC flit. A flit is made of two parts of header and payload. The header part assigned automatically by NoC and indicate the flit type and the assigned virtual channel number to the packet. The payload is 32 bit wide and will be assigned by software. The payload in buddy and tail flits contains the data which is going to be transferred to the destination core. The payload of header flit contains information for forwarding the packet between cores which has been described in Table 1.1.

1.6.1 NoC network interface (NI)

The NI works as a bridge between the wishbone bus and the NoC router. The NI transfer data between the NoC and the program RAM in a DMA fashion way. The NI has only three memory mapped registers: read, write and status register. Figure 1.2 illustrates the NI internal registers.

Table 1.1: The header flit fields description

Filed name	Description
Output port	The result of look ahead routing algorithm. It will be updated by NI at source and NoC routers along the path.
Destination x & y	The destination IP x & y address
Source x & y	The source IP address, it will be used by the shared memory IP core to send the requested data or ack to the source core
RAM_ack_req	This filed is for communicating with shared RAM. By setting this filed the processor ask for receiving an acknowledge packet from shared ram core after the write process on ram is finished
RAM_WR_req	This filed is for communicating with shared RAM. If its set the packet contains data which must be written on shared RAM otherwise it contain a read request.

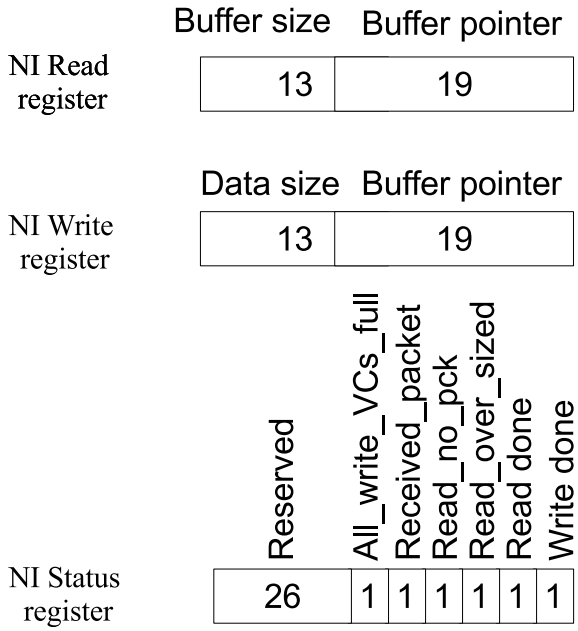


Figure 1.2: The NI registers

The Read register is used for capturing received packet from NI to the programming memory. The capturing process is started by writing the address pointer of the buffer and its size on the read register.

By writing the buffer pointer address and the size of data to be sent in word on write register the NI start sending the packet to NoC.

The status registers hold some information about the status of NI. These information are demonstrated in Table 1.6.1.

Filed name	Description
Write done	A flag which has been asserted when the packet is sent out from the NI. This flag remain zero during sending process
Read done	A flag which has been asserted when the process of transferring a packet from NI to the programming RAM is finished. This flag remain zero during transferring of packet
Read_over_size	An error flag which indicate that the length of received packet was more than the dedicated buffer size
read_no_pck	An error flag which indicate an store (read) command has been received by NI while there was no received packet in NI
All Write VCs full	This flag is asserted when all write VC are full, Hence, until this flag is asserted no packet can be sent out to the NoC

CHAPTER 2

PROJECT FILES DESCRIPTION

This chapter describe the project design files located in `/trunk/noc_based_mpsoc/src` folder. All project RTL code has been written using Verilog language and are stored in three main folders:

- **NoC:**
  This folder contains all the Verilog files required for implementation the NoC. The NoC router top level entity is located in `router.v` file.

Table 2.1: The NoC router files

File name	Description
route_compute.v	Contains the route computation modules: XY, minimal and look ahead routing modules
ext_ram_nic.v	The network interface module for sharing a memory between all cores.
sdram_core.v	The IP core including the SDRAM memory controller and ext_ram_nic.v module
switch_in.v	The input port module. This module includes the input queue buffers, request masking module and necessarily components for serving the received packets
sw_sep_alloc.v	The separable input first switch allocator.
sw_out.v	Connect the output of crossbar to the output of switch. The main propose of output switch is to hold the status of output virtual channels which in this design is shifted to the OVC_status module
sw_alloc_second_arbiter.v	The second arbitration stage in switch allocator between the input ports which request to have access to the same output port
sw_alloc_first_arbiter.v	The first stage arbitration stage in switch allocator between the input port virtual channels
router.v	The toplevel module of the NoC router, contains the input ports, crossbar, allocator and status module
ovc_status.v	This module hold the status of output virtual channels
ni.v	The wishbone adaptable NoC interface module
MUX.v	Contains some simple modules such as different type of multiplexers(one hot & conventional), bcd to one hot and ... etc.
min_number.v	The verilog module for selecting the input VC which contains the minimum number of packets among other
mask.v	The input port request masking module
fwft_fifo.v	The small first word fall through FIFO
fifo_buffer.v	The input port FIFO buffer
dual_port_ram.v	The simple dual port ram
Dimitrakopoulos_arbiter.v	Fast arbiters for on-chip network switches developed by Dimitrakopoulos [4]
cross_bar.v	The crossbar switch
arbiter.v	The top level arbiter contain bcd and one hot key arbiter
tasks.v	Contains simulation tasks for packet injection and sinking to NoC

- **IP**

  This folder contain the Verilog files required for generating the processing tiles

File name	Decribtion
wishbone_bus.v	The parametrizable wishbone bus generator
signal_holder.v	A simple counter for holding the input signal after reseting
prog_ram.v	The aeMB programing ram. The ram can be accessed via Altera in system memory content editor
gpio.v	The parameterizable general purpose input output module
altera_reset_synchronizer.v	Altera reset synchronizer
aeMB_IP.v	The aeMB SoC top module contains one aeMB processor, programming ram and optional gpio, time or external interrupts module
aeMB_mpsoc.v	The top level aeMB noc based mpsoc module
bus_addr_cmp.v	The wishbone bus peripheral devices address comparator module
reset_jtag.v	A reset signal controlled by Altera in system source and probe editor. It is used for keeping all the processor in reset mode during programming time
int_ctrl.v	The intrrupt controller module
ext_int.v	The external intrrupt module
timer.v	A simple 32 bit timer with IRQ
sdram/synthesis/submodules/ sdram_up_clocks_0.v	Altera sdram controller module
sdram/synthesis/submodules/ sdram_sdram_controller.v	Altera sdram controller module
sdram/synthesis/submodules/ Altera_reset_controller.v	Altera sdram controller module
sdram/synthesis/sdram.v	Altera sdram controller module

- **aeMB**

  This folder contains the source code of aeMB processor. The only modification which has been done on original aeMB processor is adding the valid flags to the cache memory data.

REFERENCES

1. Dally, W. and Towles, B. Route packets, not wires: on-chip interconnection networks. *Proceedings on Design Automation Conference*. 2001. 684–689.

2. Ngiap, S. T. S. AEMB 32-bit Microprocessor Core Datasheet, 2007.

3. Altera. DE2-115 Development and Education Board. http://www.altera.com/education/univ/materials/boards/de2-115/unv-de2-115-board.html.

4. Dimitrakopoulos, G., Chrysos, N. and Galanopoulos, K. Fast arbiters for on-chip network switches. *International Conference on Computer Design ICCD*. IEEE. 2008. 664–670.