XAPP745 (v1.0) September 4, 2012

Processor Control of Vivado HLS Designs

Author: Fernando Martinez Vallina

Summary

This application note describes how IP blocks generated with the Vivado™ High-Level Synthesis tool can be controlled from a processor within the Zynq™-7000 All Programmable SoC. The techniques described in this application note are also applicable to MicroBlaze™ processor-based systems.

Introduction

Vivado synthesis provides a tool and methodology for migrating algorithms from a processor onto the programmable logic. In the context of the Zynq-7000 All Programmable SoC, this means moving code from the ARM® dual-core Cortex™-A9 processor to the programmable logic for acceleration. The code implemented with Vivado synthesis in hardware represents the computational bottleneck of the algorithm. This bottleneck can be discovered through code profiling. See *EDK Concepts, Tools, and Techniques: A Hands-On Guide to Effective Embedded System Design* (UG683) for instructions on how to profile processor code.

The focus of this application note is the effective processor control of IP blocks generated with the Vivado High-Level Synthesis (HLS) tool. Aspects to be discussed regarding the use of IP blocks generated with the Vivado HLS tool in a program for the Zynq-7000 All Programmable SoC are:

- Vivado HLS tool signal level protocols and automatically generated IP-specific application program interfaces (API)
- Interrupt generation and a basic interrupt service routine (ISR) for the Zynq-7000 All Programmable SoC
- Basic program structure for accessing Vivado HLS tool IP blocks from the processor

Programming Environment Specifics

This application note assumes that the user has some general knowledge of the Vivado HLS and XPS tools. For more information on these tools see *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) and *EDK Concepts, Tools, and Techniques: A Hands-On Guide to Effective Embedded System Design* (UG683).

Signal Level Protocols

The Vivado HLS tool supports three general categories of signals: streaming interfaces, BRAM interfaces, and scalar I/O interfaces. From these categories, only the scalar I/O interfaces are accessible from the processor over the AXI4-Lite interface. Therefore, the automatically generated IP block APIs from the Vivado HLS tool are only available for IP block status signals and user-specified scalar I/O ports. A complete description of available scalar I/O protocols can be found in *Vivado Design Suite User Guide: High-Level Synthesis* (UG902). Table 1 shows the mapping between user C/C++ level scalar I/O, the Vivado HLS tool I/O protocols and the AXI4-Lite interface.

Table 1: **Vivado HLS Tool I/O Protocol Mapping to AXI4-Lite**

Processor Interface	**Argument Type**	**Variable**			**Pointer Variable**			**Array**			**Reference Variable**		
AXI4-Lite		**Pass-by-Value**			**Pass-by-Reference**			**Pass-by-Reference**			**Pass-by-Reference**		
Slave	**Interface Type**	**I**	**I/O**	**O**	**I**	**I/O**	**O**	**I**	**I/O**	**O**	**I**	**I/O**	**O**
	ap_none	D	▨	▨	D			▨	▨	▨	D		
	ap_stable		▨	▨			▨	▨	▨	▨			▨
	ap_ack		▨	▨				▨	▨	▨			
	ap_vld		▨	▨			D	▨	▨	▨			D
	ap_hs		▨	▨					▨				
	ap_ctrl_hs	▨	▨	D	▨	▨	▨	▨	▨	▨	▨	▨	▨

▨ = Unsupported interface   □ = Supported interface

From the protocols shown in Table 1, the ap_ctrl_hs protocol is for IP block start/stop and status monitoring. In terms of mapping to the AXI4-Lite bus, the start (ap_start), idle (ap_idle), and done (ap_done) signals of the IP block are mapped into registers of the interface. These signals exist by default for all designs generated by the Vivado HLS tool.

For any Vivado HLS tool design with scalar I/O ports, the design can have as many independent AXI4-Lite interfaces as I/O ports. It is also possible to group all scalar and IP block control ports into a single AXI4-Lite interface. It is recommended for the user to create a single AXI4-Lite interface per Vivado HLS tool design. This simplifies the physical connection in the XPS tool as well as the memory space allocation in the processor memory map. Instructions on how to create AXI interfaces with the Vivado HLS tool are available in *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

The Vivado HLS tool port protocols, which can be mapped to the AXI4-Lite bus, can be categorized as either raw or qualified I/O protocols. Table 2 shows how the Vivado HLS tool protocols for scalar I/O interfaces are categorized.

Table 2: **Vivado HLS Tool I/O Protocol Classification**

Raw I/O Protocol	**Qualified I/O Protocol**
ap_none	ap_hs
ap_stable	ap_ack
	ap_vld

The categorization of Table 2 has a direct impact on how the processor software is written.

Vivado HLS Tool IP-Specific API

For every IP block generated by the Vivado HLS tool, a complimentary API is automatically created to enable software development for the processor. The files describing the API are stored in the `include` directory of the PCore for use by the XPS tool. Detailed information on how to generate a PCore with the Vivado HLS tool can be found in *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Example Function in the Vivado HLS Tool

The following C function is used in the explanation of the generated API:

```
int example (int A, int B)
{
         int result;
         result = A * A + B;
         return result;
}
```

The API files generated by the Vivado HLS tool have this naming convention:

`X`{*top level function name*}_{*AXI4-Lite interface name*}`.h`

`X`{*top level function name*}`.h`

The first file defines the address map of the IP block. All ports mapped to the AXI4-Lite interface are treated as registers. The addresses of these registers are automatically assigned and are not under user control. In regards to system integration, the generated address map is a list of register offsets from the base address determined by the system architect. In the case of systems without an operating system, also referred to as bare metal systems, this file can be ignored. The Vivado HLS tool provides a header file with all of the necessary driver functions for bare metal software development. Linux drivers are currently not generated by the Vivado HLS tool. If a Linux driver is required, development of such a driver can be started from the address map file.

As previously mentioned, the second file contains all the driver functions needed for bare metal software development. The functions provided in this file assist with these operations:

- IP Block Initialization and Status Monitoring
- I/O Read and Write Function
- Interrupt Handling

The API functions provided by the Vivado HLS tool have this naming convention:

`X`{*top level function name*}_{*operation*}

IP Block Initialization and Status Monitoring

There are four standard functions for every Vivado HLS tool IP block:

- `X`{*top level function name*}`_Initialize()`
- `X`{*top level function name*}`_Start()`
- `X`{*top level function name*}`_IsDone()`
- `X`{*top level function name*}`_IsIdle()`

Using the code example from the Example Function in the Vivado HLS Tool, the required functions in any processor program are:

```
XExample_Initialize()
XExample_Start()
```

The initialization function `XExample_Initialize` has no effect on the IP block at the hardware level. The purpose of this function is to store identifier information for a specific instance of the example function in hardware. This identifier information is required by all other driver functions to ensure communication with the correct hardware module. The designer can use the same API to communicate with as many fabric instantiations of a hardware accelerator as needed by the application.

The start function, `XExample_Start`, pulses the `ap_start` signal on the target Vivado HLS tool hardware accelerator. This is a 1 clock cycle pulse that initiates the operation of the IP block. Depending on the Vivado HLS tool protocols selected for the function I/O, the location of

XExample_Start in the processor code must be changed to guarantee proper execution of the accelerator. Ports with a raw I/O protocol such as ap_none and ap_stable must be written before the start signal is received by the accelerator. Ports with qualified I/O protocols can be written before or after the XExample_Start function as long as the sequencing of the protocol is respected.

The other two functions generated for the example IP block are:

```
XExample_IsDone()
XExample_IsIdle()
```

Both of these functions are optional and allow the processor to monitor the status of the IP block.

XExample_IsDone can be used by the processor to check when the IP block started by XExample_Start has finished execution. This enables poling the IP block for task completion.

XExample_IsIdle tells the processor if the core is running or not. It is not a deadlock checking function. This function returns *false* from the time the start function is executed by the processor until the ap_done signal is asserted by the IP block. If the system causes the Vivado HLS tool IP block to deadlock, the *_IsIdle function also returns *false*.

I/O Read and Write Function

The number of I/O read and write functions directly corresponds to the number of I/O ports in the IP block. From the Example Function in the Vivado HLS Tool, page 3, the function signature of the IP block generated with the Vivado HLS tool is:

```
int example(int A, int B)
```

This function signature states that there are 2 input ports A and B and 1 output port for the function return value. Also, assume that port A has I/O protocol ap_none and port B has I/O protocol ap_hs. In this case, port A is associated with these driver functions:

```
XExample_SetA()
XExample_GetA()
```

The function XExample_SetA writes a value from the processor to port A of the IP block. As a result of using the ap_none protocol on port A, this function must be executed by the processor before the XExample_Start function. All ports utilizing the ap_none protocol are sampled and registered by the IP block at the time ap_start signal is received. Ports using the ap_stable protocol must also be set before the start signal is issued. Furthermore, the value of these ports must remain constant during the execution of the IP block. Ports with the ap_stable I/O protocol are never registered by the Vivado HLS tool IP block.

The purpose of the get function, XExample_GetA, is to verify the value written from the processor to the IP block.

As a result of using a qualified I/O protocol, port B is associated with these driver functions:

```
XExample_GetB()
XExample_SetB()
XExample_SetB_Vld()
```

The function `XExample_SetB()` behaves in the same manner as in the case of port A. This is a data transfer function. The difference is in the requirement to also use the `XExample_SetB_Vld` function. This function, which is a direct result of the I/O protocol of port B, tells the IP block when the value of port B is valid and can be registered into the core. The evaluation of the valid signal on port B does not occur until the IP block receives the start signal from the processor. The processor can write the value of B at any time within its program execution. The Vivado HLS tool IP block stalls and waits for the valid signal in port B before continuing operation.

The read functions return the value of a register in the Vivado HLS tool IP block to the processor. In the case of input ports A and B, the read functions are

```
XExample_GetA()
XExample_GetB()
```

The purpose of these functions is to read the contents of a register in the Vivado HLS tool IP block. All user defined I/O ports have a read function associated with them. In the case of the function return value, the read function is of the form:

```
XExample_GetReturn()
```

The value of `XExample_GetReturn` is only valid after the done signal has been received from the IP block. Depending on how the processor software is written, the done signal from the IP block can be captured by polling or with an Interrupt Service Routine (ISR).

Interrupt Handling

IP blocks created with the Vivado HLS tool can generate two kinds of interrupts:

1. Task completion
2. Task pipelining

The task completion interrupt occurs at the end of a function call synthesized into hardware. This is the done signal (ap_done) asserted by the IP block.

Task pipelining allows the processor to launch multiple calls to the same hardware accelerator before the previous call is finished. This reduces computation latency and reduces the number of accelerators required to achieve a system performance specification. In addition to supporting multiple simultaneous calls from the processor, IP blocks with task pipelining enabled can handle different configuration parameters for each call. From the perspective of the processor, the difference between having or not having task pipelining is shown in Figure 1.

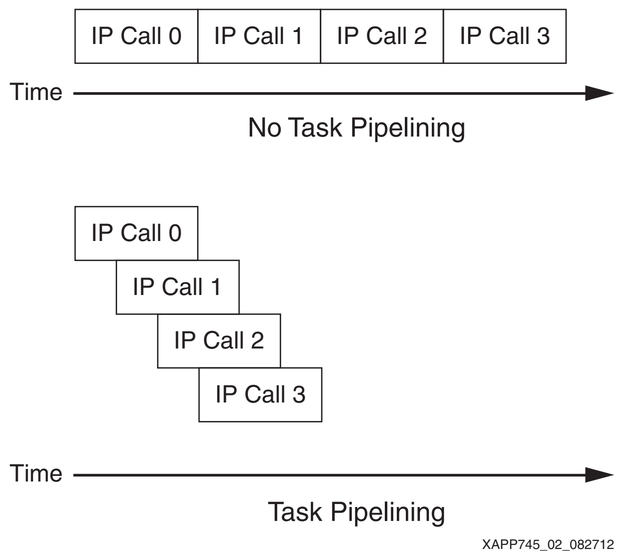

Figure 1: **Task Pipelining vs No Task Pipelining**

The interrupt APIs for Vivado HLS tool IP blocks provide support for both done signal and task pipelining interrupt sources. In addition, the APIs have been designed with support for up to 32 interrupt sources per IP block.This capability will be used in future versions of the Vivado HLS tool.

As far as the processor is concerned, each Vivado HLS tool IP block acts as a single interrupt source. Once the interrupt is received, the interrupt service routine (ISR) must access the interrupt logic in the IP block to determine the cause. Issuing an interrupt to the processor is controlled by two functions:

```
XExample_InterruptGlobalEnable()
XExample_InterruptGlobalDisable()
```

Along with these two functions, there are APIs to control the behavior of individual interrupt sources within the IP block. Interrupt sources are 1-hot encoded onto a 32-bit control word inside the generated IP block. Bit 0 is the least significant bit of the word and encodes the status of the done signal interrupt.

Enabling and disabling interrupt sources in the IP block is carried out by:

```
XExample_InterruptEnable(…)
XExample_InterruptDisable(…)
```

In cases where the IP block has more than one interrupt source, the processor must determine the cause of the interrupt. The cause can be determined by using these functions:

```
XExample_InterruptGetEnabled()
XExample_InterruptGetStatus()
```

The function `XExample_InterruptGetEnabled` returns a list of which sources in the IP block are allowed to issue an interrupt to the processor. This is used in conjunction with the status function, `XExample_InterruptGetStatus()`, to determine which source caused the processor interrupt. The status function returns the list of sources trying to issue an interrupt. One thing to keep in mind is that a Vivado HLS tool IP block does not have the concept of interrupt priority. Therefore, multiple internal sources can be active at the same time. Interrupt priority and handling is determined by the ISR.

Once the processor has determined the interrupt source, it must clear the source in the IP block. Clearing an interrupt is achieved using the clear function:

```
XExample_InterruptClear(…)
```

Additional information on the API syntax generated by the Vivado HLS tool can be found in *Vivado Design Suite User Guide: High-Level Synthesis* UG902.

Basic Processor Program

Any program making use of IP blocks generated by the Vivado HLS tool must execute these tasks:

1. Initialize the IP Block in the Processor Program Space
2. Have a Basic Interrupt Service Routine for the IP block
3. Initialize the Processor Exception Table and register the IP block ISR
4. Write Data to the IP Block
5. Start the IP Block
6. Process the IP Block Return Value

Initialize the IP Block in the Processor Program Space

Initializing the IP block in the processor program space requires the use of two structures provided by the Vivado HLS tool. These structures are IP-specific. In the context of the Example Function in the Vivado HLS Tool, page 3, the structures are `XExample` and `XExample_Config`.

`XExample` declares a pointer to a specific instance of the accelerator in the programmable logic. `XExample_Config` declares a struct which holds an instance ID number and the base address of the IP block from the processor memory map. Initialization is completed using the following code:

```
XExample ex;
XExample_Config ex_config = {0,XPAR_EXAMPLE_TOP_0_S_AXI_EXAMPLE_BASEADDR};

XExample_Initialize(&ex,&ex_config);
```

Basic Interrupt Service Routine

An ISR for a Vivado HLS tool generated IP block must have these elements:

- Disable IP block global interrupt
- Fetch IP block interrupt enable list
- Fetch IP block interrupt status
- Clear the interrupt

The following code shows how to create an ISR for the example function:

```
void ExampleISR(void *InstancePtr){
     int enabled_list;
     int status_list;
     XExample *pEx = (XExample *) InstancePtr;
     //Disable Global Interrupt
     XExample_InterruptGlobalDisable(pEx);
     //Get list of enabled interrupts
     enabled_list = XExample_InterruptGetEnabled(pEx);
     //Get interrupt status list
     status_list = XExample_InterruptGetStatus(pEx);
     //Check ap_done created the interrupt
     if((enabled_list & 1) && (status_list & 1)){
          //Clear the ap_done interrupt
          XExample_InterruptClear(pEx,1);
          //Set a result status flag
          NewResult = 1;
     }
}
```

Initialize the Processor Exception Table

Detailed information on how to initialize the processor exception table can be found in *EDK Concepts, Tools, and Techniques: A Hands-On Guide to Effective Embedded System Design* (UG683). The following code shows one method of initializing the processor exception table and registering the example ISR:

```
void SetupInterrupt(){
     int result;
     // Find the interrupt configuration table
     XScuGic_Config *pCfg =
             XScuGic_LookupConfig(XPAR_SCUGIC_SINGLE_DEVICE_ID);
     // Initialize the Interrupt Controller
     Result = XScuGic_CfgInitialize(&ScuGic, pCfg, pCfg->CpuBaseAddress);
     //Initialize the exception handler
     Xil_ExceptionInit();
     //Register the exception handler
     Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
             (Xil_ExceptionHandler) XScuGic_InterruptHandler,&ScuGic);
     //Enable the exception handler
     Xil_ExceptionEnable();
     //Connect the Example ISR to the exception table
     result = XScuGic_Connect( &ScuGic,
                              XPAR_EXAMPLE_INTERRUPT_INTR,
                              (Xil_InterruptHandler)ExampleISR,&ex);
     //Enable the Example ISR
     XScuGic_Enable(&ScuGic,XPAR_EXAMPLE_INTERRUPT_INTR);
     return result;
}
```

Write Data to the IP Block

For the example function, writing a value of 5 to port A and 10 to port B can be accomplished using the following code:

```
XExample_SetA(&ex,5);
XExample_SetB(&ex, 10);
XExample_SetB_Vld(&ex);
```

Start the IP Block

Starting an IP block instance from the processor involves three steps:

1. Enable interrupt sources
2. Enable global IP block interrupt
3. Issue the start signal to the IP block

The following code shows a start function for the example IP block:

```
void ExampleStart(void *InstanePtr){
       XExample *pEx = (XExample *)InstancePtr;
       //Enable ap_done as an interrupt source
       XExample_InterruptEnable(pEx,1);
       //Enable the Global IP Interrupt
       XExample_InterruptGlobalEnable(pEx);
       //Start the IP
       XExample_Start(pEx);
}
```

Process the IP Block Return Value

The IP block return value is available to the processor as shown in the following code example:

```
ip_result = XExample_GetReturn(&ex);
```

Processor Main Function

The functions described so far are necessary elements of a standalone software application for the Zynq-7000 All Programmable SoC, which makes use of one or more Vivado HLS tool generated IP blocks. The processor main function can be written as follows:

```
int main(){
        int result;
        //Initialize the IP
      XExample_Initialize(&ex,&ex_config);

      //Setup the Interrupt for the System
      SetupInterrupt();

      //Write the values for port A and B
      XExample_SetA(&ex,5);
      XExample_SetB(&ex, 10);
      XExample_SetB_Vld(&ex);

      //Start the IP
      ExampleStart(&ex);

      //Wait for the core interrupt
      while(!NewResult);

      //Get the return value of the IP
      result = XExample_GetReturn(&ex);
      printf("IP result = %d\n\r",result;

      return 0;
}
```

Conclusion

The Vivado HLS tool provides a standalone driver API for every IP block generated with the tool. This application note provides a description of how to use the generated API within a Zynq-7000 platform processor program. The fundamental programming techniques described in this document apply to all Vivado HLS tool IP blocks which communicate with a processor.

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
09/04/12	1.0	Initial draft.

Notice of Disclaimer