Matrix Multiplier PCIe Endpoint - Integration Guide

Overview

This is a simple matrix multiplication hardware accelerator that implements a PCIe endpoint device. It's designed to be much easier to understand than the CPM-QDMA demo while still demonstrating real PCIe concepts.

Device Features

- **Single BAR0**: 4KB MMIO region for register access
- **DMA Master**: Can read/write host memory
- **MSI Interrupt**: Signals completion via interrupt line
- **Simple Operation**: Configure pointers, start, wait for interrupt

Register Map (BAR0)

Offset	Name	Access	Description
0x0000	CONTROL	R/W	Control register (bit 0: START, bit 1: RESET)
0x0004	STATUS	R	Status register (IDLE/BUSY/DONE/ERROR)
0x0008	DIM_N	R/W	Matrix dimension (NxN)
0x0010	MATRIX_A_PTR	R/W	64-bit DMA pointer to Matrix A
0x0018	MATRIX_B_PTR	R/W	64-bit DMA pointer to Matrix B
0x0020	MATRIX_C_PTR	R/W	64-bit DMA pointer to Matrix C
0x0028	INT_STATUS	R/W1C	Interrupt status (write 1 to clear)
0x002C	INT_ENABLE	R/W	Interrupt enable mask

How It Works

1. **Initialization**: Driver allocates DMA buffers for matrices A, B, and C
2. **Configuration**: Driver writes matrix dimension and buffer addresses to device registers
3. **Start**: Driver sets START bit in CONTROL register
4. **Execution**: Device reads matrices A and B via DMA, computes C = A × B
5. **Completion**: Device writes result C to host memory and raises interrupt
6. **Cleanup**: Driver reads INT_STATUS, clears interrupt, reads result

Integration with Xilinx PCIe Controller

Step 1: Understand the Xilinx PCIe Controller Structure

The Xilinx PCIe repo typically has these components:

```
xilinx_pcie_ep_device  (PCIe Endpoint Controller)
  ├── Config space handling
  ├── BAR mapping
  ├── TLP packet processing
  └── Socket interfaces
```

Step 2: Key Socket Connections

Your device needs TWO socket connections:

A. BAR0 Target Socket (MMIO from Host → Device)

cpp

```cpp
// In your top-level module:
pcie_controller->bar0_initiator_socket.bind(matrix_device->bar0_target_socket);
```

This allows the PCIe controller to forward MMIO reads/writes to your device's registers.

B. DMA Initiator Socket (Device → Host Memory)

cpp

```cpp
// In your top-level module:
matrix_device->dma_initiator_socket.bind(pcie_controller->dma_target_socket);
```

This allows your device to perform DMA reads/writes to host memory.

Step 3: Configure PCIe Device Parameters

When instantiating the Xilinx PCIe device, you need to configure:

cpp

```cpp
// Pseudo-code - exact syntax depends on Xilinx implementation
pcie_device->set_vendor_id(0x10EE);      // Xilinx vendor ID
pcie_device->set_device_id(0x9038);      // Your device ID
pcie_device->set_class_code(0x118000);   // Unassigned class
pcie_device->set_bar0_size(0x1000);      // 4KB BAR0
pcie_device->set_revision_id(0x01);
```

Step 4: Interrupt Connection

```cpp
sc_signal<bool> irq_signal;

// Connect device interrupt to PCIe controller
matrix_device->interrupt(irq_signal);
pcie_controller->msi_interrupt_in(irq_signal);
```

Step 5: Example Integration Code

```cpp
SC_MODULE(complete_pcie_system) {
    // Components
    xilinx_pcie_ep_device *pcie_ctrl;
    matrix_multiplier_pcie *matrix_dev;

    // Signals
    sc_signal<bool> irq;

    SC_CTOR(complete_pcie_system) {
        // Create matrix device
        matrix_dev = new matrix_multiplier_pcie("matrix_dev");

        // Create PCIe controller
        pcie_ctrl = new xilinx_pcie_ep_device("pcie_ctrl");

        // Configure PCIe device
        pcie_ctrl->configure_device(
            0x10EE,    // Vendor ID
            0x9038,    // Device ID
            0x118000,  // Class code
            0x1000     // BAR0 size
        );

        // Connect BAR0 (Host MMIO → Device registers)
        pcie_ctrl->bar0_init_socket.bind(matrix_dev->bar0_target_socket);

        // Connect DMA (Device → Host memory)
        matrix_dev->dma_initiator_socket.bind(pcie_ctrl->dma_tgt_socket);

        // Connect interrupt
        matrix_dev->interrupt(irq);
        pcie_ctrl->irq_in(irq);
    }
};
```

Connecting to QEMU

Method 1: Remote Port Protocol (Recommended)

Xilinx has a remote-port protocol for QEMU-SystemC communication:

1. **QEMU Side**: Use `-device remote-port-pci-device`
2. **SystemC Side**: Connect via remote-port adapter
3. Communication happens over Unix sockets

Method 2: Custom Socket Interface

Create a socket bridge that:

- Accepts connections from QEMU
- Converts QEMU PCIe transactions to TLM
- Forwards to your SystemC model

Testing Strategy

Phase 1: Pure SystemC Test (Provided)

- Use the testbench I provided
- Verify register access works
- Verify DMA reads/writes work
- Verify computation is correct

Phase 2: With Xilinx PCIe Controller

- Replace test_driver with Xilinx controller
- Verify TLP packets are generated correctly
- Verify BAR decoding works

Phase 3: With QEMU

- Connect socket interface
- Boot Linux in QEMU
- Device should appear in `lspci`

Phase 4: Linux Driver Development

- Write character device driver
- Implement `open()`, `close()`, `ioctl()`
- Test matrix multiplication from userspace

Compilation Instructions

```bash
# Compile with SystemC
g++ -std=c++11 \
    -I$SYSTEMC_HOME/include \
    -L$SYSTEMC_HOME/lib-linux64 \
    -o pcie_test \
    testbench.cpp \
    -lsystemc -lm

# Run
export LD_LIBRARY_PATH=$SYSTEMC_HOME/lib-linux64:$LD_LIBRARY_PATH
./pcie_test
```

Next Steps for You

1. **First**: Run the standalone testbench to understand the device behavior
2. **Second**: Look at the Xilinx PCIe examples to find:
   - How to instantiate `xilinx_pcie_ep_device`
   - What the socket names are
   - How to configure device parameters
3. **Third**: Create integration file connecting your device to Xilinx controller
4. **Fourth**: Add socket interface for QEMU connection
5. **Finally**: Start developing your Linux driver

Common Issues and Solutions

Issue: Sockets not binding

Solution: Check socket names exactly match - TLM is case-sensitive

Issue: DMA transactions failing

Solution: Ensure address ranges are within host memory bounds

Issue: Interrupts not working

Solution: Verify interrupt line is connected and INT_ENABLE register is set

Issue: QEMU can't see device

Solution: Check PCIe config space is properly initialized with valid vendor/device IDs

Device Driver Skeleton (For Reference)

Here's what your Linux driver will need to do:

c

```c
// Pseudo-code for Linux driver

static int matrix_probe(struct pci_dev *pdev, const struct pci_device_id *id) {
    // Enable device
    pci_enable_device(pdev);
    pci_set_master(pdev);

    // Map BAR0
    void __iomem *bar0 = pci_iomap(pdev, 0, 0x1000);

    // Allocate DMA buffers
    dma_addr_t dma_a = dma_alloc_coherent(&pdev->dev, size, ...);
    dma_addr_t dma_b = dma_alloc_coherent(&pdev->dev, size, ...);
    dma_addr_t dma_c = dma_alloc_coherent(&pdev->dev, size, ...);

    // Request IRQ
    request_irq(pdev->irq, matrix_isr, IRQF_SHARED, "matrix", dev);

    return 0;
}

static long matrix_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
    switch(cmd) {
    case IOCTL_MULTIPLY:
        // Write dimension
        iowrite32(n, bar0 + REG_DIM_N);

        // Write pointers
        iowrite32(dma_a, bar0 + REG_MATRIX_A_PTR);
        iowrite32(dma_b, bar0 + REG_MATRIX_B_PTR);
        iowrite32(dma_c, bar0 + REG_MATRIX_C_PTR);

        // Enable interrupt
        iowrite32(0x1, bar0 + REG_INT_ENABLE);

        // Start computation
        iowrite32(0x1, bar0 + REG_CONTROL);

        // Wait for completion
        wait_for_completion(&dev->completion);
        break;
    }
    return 0;
}

static irqreturn_t matrix_isr(int irq, void *dev_id) {
    // Read interrupt status
```

```
    u32 status = ioread32(bar0 + REG_INT_STATUS);

    // Clear interrupt
    iowrite32(status, bar0 + REG_INT_STATUS);

    // Signal completion
    complete(&dev->completion);

    return IRQ_HANDLED;
}
```

Questions?

This device is intentionally simple to help you learn. Once you understand this, you can add:

- Multiple BARs
- More sophisticated DMA engines
- Streaming data interfaces
- Power management
- Advanced interrupt mechanisms (MSI-X)

Good luck with your project!