Kapitel 5

Debugging

In diesem Teil des Projektes sollen die grundlegenden Techniken des Debuggings von auf einem SOPC basierenden Nios II System betrachtet werden. Um auf der Hardware-Ebene den Signalablauf innnerhalb selbsterstellter Komponenten analysieren zu können, kommt ein in die Quartus II Umgebung integrierter Logic-Analyser zum Einsatz. Die Softwarefunktionalität sowie die Interaktion mit der Hardware über Memory-Mapped-IO lässt sich mittels eines in der Nios II IDE vorhandenen auf dem GDB basierenden Debugger überprüfen.

5.1 Hardware-Debugging

Der in die Quartus II Umgebung integrierte SignalTap II ist ein Logic-Analyser, welcher auf einem FPGA als Logik instantiiert werden kann. Als Speicher für die anfallenden Daten, auch Samples genannt, dient dabei der freie Teil des On-Board-RAM. Die Konfiguration sowie das Auslesen der Daten erfolgt über die JTAG-Schnittstelle, so dass die Signalverläufe schließlich im Quartus II visualisiert werden können.

Der direkte Zugriff auf in der Hardware vorhandene Bitmuster bietet insbesondere dann einen großen Vorteil, wenn externe Hardware angebunden ist, die nicht mit einer HDL spezifiziert wurde und daher von einer Simulation nicht erfasst werden kann. Aber auch bei simulierbaren Systemen kann die Verwendung des Logic-Analyzers Vorteile mit sich bringen, etwa dann, wenn nur ganz bestimmte Aspekte

betrachtet werden sollen, und eine zeitgenaue Simulation gerade in Verbindung mit einem Softcore-Prozessor einen hohen zeitlichen Aufwand bedeuten würde.

Dieser mitgelieferte interne Logic-Analyser besitzt einige komplexe Funktionen, die sonst nur bei aufwendigen externen Analysern zu finden sind, beispielsweise mehrstufig verkettete Trigger, die erst dann die Aufzeichnung der Analysedaten starten, wenn mehrere Bedingungen auf gleichen oder unterschiedlichen Signalen zutreffen. Darüber hinaus lassen sich mit Hilfe von State-Machines einfache Protokolle nachbilden und zur Triggerung verwenden. Weiterhin werden auch externe Trigger unterstützt. Der Sample-Speicher lässt sich segmentieren, so dass Daten an mehreren Trigger-Zeitpunkten, die weiter auseinander liegen, aufgezeichnet werden können. Der größte Unterschied im Vergleich zu eigenständigen Logic-Analysern liegt in der starken Größenbeschränkung des Sample-Speichers, die in der Verwendung der im FPGA integrierten Speicherblöcke begründet ist.

Eine genaue Beschreibung der Funktionsweise sowie Hinweise zur Konfiguration finden Sie in Volume 3, Sektion IV, Kapitel 14 des Quartus II Handbuchs unter dem Titel „Design Debugging Using the SignalTap II Embedded Logic Analyzer“. Der Dateiname für dieses Kapitel lautet z. Z. `qts_qii53009.pdf`.

Aufgabe 1

1. Fügen Sie zu Ihrem System eine Instanz des Signal-Tap II Logic-Analysers hinzu. Machen Sie sich mit der Konfiguration und Bedienung des Werkzeugs vertraut.
2. Konfigurieren Sie den Trigger auf einen Lesezugriff auf eine per Avalon angebundene SOPC-Komponente und vergleichen Sie die erfassten Daten mit den Werten, welche die Software aus diesem Zugriff erhält. Dazu sollten Sie ein kleines Testprogramm erstellen, welches die Werte einliest und auf der Konsole ausgibt.

5.2 Software-Debugging

Das Debugging der Software basiert auf dem zum GCC gehörenden GNU-Debugger GDB und ist in das NIOS II IDE integriert, wobei die Grundfunktionalität der Bedienoberfläche bereits vom Eclipse-Framework bereitgestellt wird. Die Debugging-Ansicht des IDE dient somit als GUI für den GDB, und beherrscht die übliche vom Debugger bereitgestellte Funktionalität.

Der Ablauf von Programmen kann unterbrochen und fortgesetzt werden, und es kann ein Zugriff auf die Systemparameter erfolgen. So lassen sich z. B. der Inhalt von Registern, Variablen, sowie des gesamten Speicherbereichs auslesen und verändern. Mittels Stepping lassen sich Programme Schritt für Schritt sowohl im C-Code als auch im zugehörigen Assembler-Code ausführen. Es können also die Auswirkungen einzelner Befehle genau beobachtet werden. Über so genannte Breakpoints lässt sich ein laufendes Programm automatisch stoppen, wenn während der Ausführung bestimmte, zur Laufzeit definierbare Stellen im Code erreicht werden.

Aufgabe 1

Starten Sie für eines Ihrer vorhandenen Programme den Debugging-Modus, in dem Sie statt dem üblichen „Run as...“ das „Debug as...“ verwenden.

1. Führen Sie zunächst eine schrittweise Abarbeitung ihres C-Codes durch und beachten Sie die Auswirkungen auf die Register und Variablen. Experimentieren Sie mit den unterschiedlichen Schritt-Möglichkeiten („Step into“, „Step over“ „Step out“).
2. Schalten Sie den Schrittmodus auf Assembler-Instruktionen um. Vergleichen Sie den Assembler-Code mit Ihrem C-Code.
3. Fügen Sie in der Mitte Ihres Codes, z. B. innerhalb einer großen Schleife, einen Breakpoint ein. Lassen Sie das Programm bis zu diesem Punkt durchlaufen.
4. Schauen Sie sich bekannte Adressbereiche (z. B. den Ihrer eigenen Avalon-Komponente; die Adresse können Sie der Datei `system.h` oder der Übersicht

im SOPC-Builder entnehmen) im Speicher an. Beobachten Sie, wie Ihre Treiberfunktionen die in den Speicher eingeblendeten Register verändern.

5. Schreiben Sie direkt mittels des Debuggers in die Register Ihrer Komponente. Beobachten Sie die Auswirkungen auf die Hardware.

Beachten Sie dabei, dass der direkte schreibende Zugriff aus dem Debugger momentan nur für 8 Bit breite Register (also die Ihrer ersten Implementierung) funktioniert. Bei breiteren Registern verwendet der Debugger einen Byte-weisen Zugriff, der nur dann zu korrekten Ergebnissen führt, wenn Ihre Komponente über das `byteenable`-Signal des Avalon-MM verfügt und dieses korrekt interpretiert. Der Zugriff aus Ihrem C-Code sollte hingegen mit 32 Bit breiten Zugriffen arbeiten, so dass er nicht von dieser Einschränkung betroffen ist. Falls Ihre Komponente Register verwendet, die breiter als 8 Bit sind, können Sie durch nachträgliche Implementierung der `byteenable`-Funktionalität (siehe Avlalon-Spezifikation) eine korrekte Behandlung der Schreibzugriffe des Debuggers erreichen.