Und wieder bockt der Multicore

Hilfsmittel und Werkzeuge für das Multicore-Debugging

Jens Braunes, PLS Programmierbare Logik & Systeme GmbH

Beim Debugging tief eingebetteter Multicore-Systeme geht es nicht mehr nur um das Aufspüren von falschen Variablenwerten, auf der Tagesordnung stehen vielmehr Deadlocks, Ressourcenkonflikte oder Timing-Probleme. Für den Entwickler bedeutet das eine große Herausforderung, die nur durch geeignete Unterstützung durch On-Chip-Debug-Funktionen im engen Zusammenspiel mit leistungsfähigen Softwarewerkzeugen gemeistert werden kann. Der folgende Beitrag stellt solche Lösungen vor und zeigt ihre Möglichkeiten, aber auch ihre Grenzen.

Wenn man nur den Consumer-Markt betrachtet, sind Multicore-Systeme schon seit zehn Jahren im Mainstream angekommen. Bei den tief eingebetteten Systemen hingegen, wie wir sie vor allem in Industrieanwendungen oder bei Automotive-Steuerungen finden, wurde der Technologiewechsel erst in den letzten Jahren vollzogen, und das auch recht zögerlich. Ein Grund dafür sind sicherlich die hohen Anforderungen an Sicherheit, Zuverlässigkeit und Echtzeit, die in den genannten Bereichen nun einmal absoluten Vorrang genießen. Aber auch das umfangreiche Portfolio an bewährten und gut getesteten Softwaremodulen für Singlecore-Systeme, deren Portierung auf mehrere Kerne einen nicht zu unterschätzenden Aufwand bedeutete, bremste hier sicherlich einen schnelleren Vormarsch aus. Und nicht zu vergessen:

Multicore ist nicht gleich Multicore

Beim aus der Windows-, Linux- und Android-Welt bekannten Multicore-Ansatz können Tasks und Prozesse dynamisch erzeugt werden, die dann je nach Last auf einem beliebigen Kern zur Ausführung kommen. Das ist möglich, weil die verwendeten Prozessoren über identische Rechenkerne verfügen, es sich also um homogene Multicore-Systeme handelt. Jeder Kern ist in der Lage und auch geeignet, zugewiesene Tasks gleichermaßen auszuführen.

Industrial- und Automotive-Anwendungen erfordern allerdings in der Regel festgelegte Abarbeitungszeiten für einzelne Tasks und garantierte Antwortzeiten. Deshalb kommen hier zumeist heterogene Multicore-Systemen mit mehreren speziell auf eine bestimmte Aufgabe zugeschnittenen Kernen zum Einsatz, wie das Beispiel der AURIX™-Mikrocontroller aus dem Hause Infineon verdeutlicht (**Fehler! Verweisquelle konnte nicht gefunden werden.**). Zwar stammen die drei Hauptkerne allesamt aus der TriCore-Architekturfamilie, jedoch sind nur zwei davon als sogenannte Performancekerne (P-Cores) für normale rechenintensive Aufgaben vorgesehen. Die dritte, als Economy-Kern (E-Core) bezeichnete Zentraleinheit übernimmt vorrangig die Verwaltung der Peripherals und allgemeine, weniger Rechenleistung erforderliche Aufgaben. Für sicherheitskritische Tasks wurde einer der P-Cores und der E-Core mit einem zusätzlichen Lockstep-Core ausgestattet, der im Hintergrund die gleichen Operationen wie der eigentliche Kern ausführt. Tritt beim Vergleich der Ergebnisse eine Abweichung auf, so arbeitet das System nicht mehr zuverlässig. Es muss gegebenenfalls in einen sicheren Zustand zurückversetzt werden.

Komplexe Zeitsteueralgorithmen sowie die effiziente und parallele Verarbeitung von

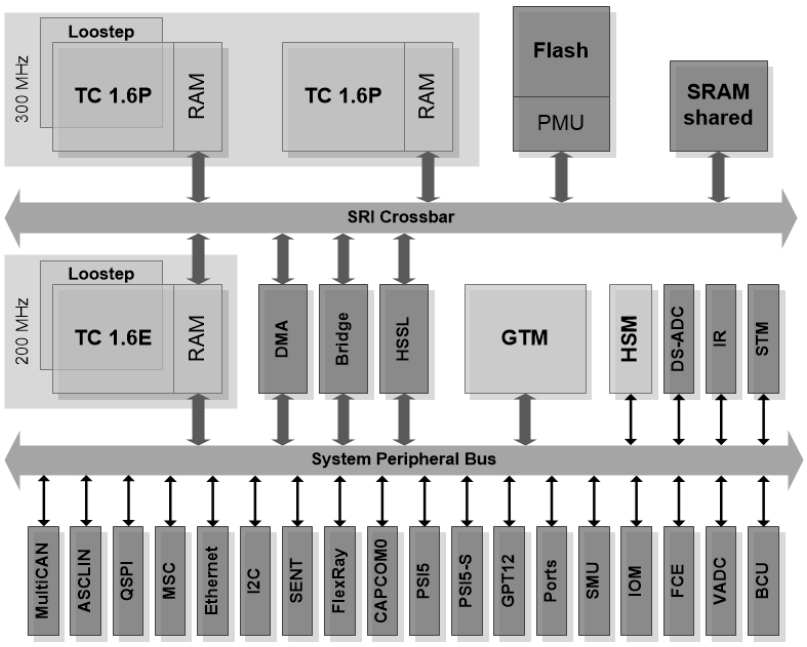

Abbildung 1: AURIX™ Multicore-Architektur von Infineon

Signalen unterstützt ein vierter „Kern“, das sogenannte Generic Timer Module (GTM). Völlig andersartig als die TriCore-Kerne, ist das GTM dennoch über einen eigenen Befehlssatz programmierbar. Dadurch können auch Tasks darauf ausgeführt werden.

Wenn es also um die Verteilung der Applikationslasten auf die Rechenkerne geht, kann man hier kaum auf das Betriebssystem vertrauen. Vielmehr muss bereits während des Softwareentwurfs klar sein, welcher Kern welche Aufgaben zu übernehmen hat.

Tief eingebettetes Debugging

Vor allem Applikationen, die ihre einzelnen Aufgaben mit hohen Echtzeitanforderungen und verteilt auf unterschiedliche Rechenkerne abarbeiten, stellen für Debugging, Test und Systemanalyse oftmals eine große Herausforderung dar. So wirken sich die im Normalfall zumeist ziemlich großen Abhängigkeiten zwischen den auf unterschiedlichen Kernen ausgeführten Tasks natürlich auf das Stop-Go-Debugging aus. Man kann nicht einfach unbedacht einen Kern anhalten, während alle anderen weiterlaufen. Mitunter müssen auch die restlichen Kerne und die Peripherals gleichzeitig angehalten werden, damit die Applikation nicht gänzlich außer Tritt und in einen undefinierten Zustand gerät. Ein wirklich gleichzeitiges Anhalten ist aber bei heterogenen Kernen mit unterschiedlicher Taktung und Ausführungs-Pipelines kaum möglich. Deshalb wird es in der Praxis immer einen gewissen Zeitversatz geben, mit dem man als Entwickler leben muss. Manchmal kann das Anhalten eines kompletten Multicore-Systems sogar fatale Folgen haben, zum Beispiel, wenn parallel noch andere Anwendungen laufen, die zu diesen Zeitpunkt nicht debuggt werden sollen oder dürfen. Die genannten Anwendungsszenarien machen deutlich, wie wichtig ein flexibles, synchrones Run-Control für die Multicore-Debug-Infrastruktur ist,

Ein zweiter, bedeutsamer Aspekt ist die Analyse des Laufzeitverhaltens, und zwar ohne dabei selbiges zu beeinflussen. Diese nicht-intrusive Systembeobachtung spielt

nicht nur bei echtzeitkritischen Anwendungen, sondern auch bei Profiling-Aufgaben oder für die Beobachtung der Kommunikation zwischen den Kernen eine wichtige Rolle. Oftmals ist es wünschenswert, den jeweiligen Systemzustand zu einem bestimmten Zeitpunkt mit Hilfe des extern angeschlossenen Debuggers aus dem Target-System auslesen zu können. Allerdings würde ein Anhalten der Applikation das Systemverhalten unter Umständen so grundlegend verändern, dass es nichts mehr mit dem ohne angeschlossenen Debugger zu tun hätte. Daraus folgt: Für eine effiziente nicht-intrusive Systembeobachtung ist Tracen unerlässlich.

On-Chip Debugger

Doch zuerst noch einmal zurück zum Thema synchrones Run-Control. Hierfür sind schnelle Signalwege zwischen den Kernen notwendig, die sich nur mit Debug-Hardware direkt auf dem Chip realisieren lassen. Stop- und Go-Signale von extern über die Debug-Schnittstelle zu übermitteln, würde bei den heutzutage üblichen hohen Taktfrequenzen viel zu lange dauern. Die Applikation käme unweigerlich außer Tritt.

Nun bietet jeder Chip-Hersteller seine eigene On-Chip Debug-Lösung an. Infineon beispielsweise nennt sie OCDS (On-Chip Debug System). Ein wichtiger Bestandteil darin ist ein Trigger-Switch, der Halt- und Suspend-Signale einzeln konfigurierbar systemweit verteilt. Damit lassen sich einzelne Rechenkerne und Peripherieeinheiten ohne Beeinflussung der restlichen Funktionsgruppen ganz gezielt gleichzeitig anhalten und wieder starten. Zusätzlich können einzelne Trigger-Leitungen des Trigger-Switches auch über Pins nach außen geführt werden. Das ist eine interessante Option, um beispielsweise ein Oszilloskop anzuschließen, oder gar ein Break von außerhalb des Chips auszulösen.

Neben der AURIX-Familie von Infineon gibt es natürlich noch eine Reihe anderer Multicore-Mikrocontroller, die den Industrial- und Automotive Sektor abdecken, darunter beispielsweise Freescales MPC57xx-Familie oder SoCs auf Basis der ARM-Cortex-R-Architektur. Werfen wir als erstes einen Blick auf CoreSight™ [1], der On-Chip-Debug-Hardware von ARM.

Für die Verteilung der Break- und Go-Signale zwischen den Kernen wird eine sogenannte Cross-Trigger-Matrix (CTM) mit daran angeschlossenen Cross-Trigger-Interfaces (CTI) genutzt (**Fehler! Verweisquelle konnte nicht gefunden werden.**). Kanäle in der CTM leiten die Signale im Broadcast-Modus an die angeschlossenen CTIs weiter. Diese sind wiederum direkt mit den Kernen verbunden und so konfigurierbar, dass sie die Signale für das Run-Control zwischen Kern und CTM wahlweise entweder weiterleiten oder blockieren. Aufgrund der notwendigen Handshake-Mechanismen zwischen den beteiligten Komponenten kommt es dabei zu Signalverzögerungen von mehreren Takten. Wie groß diese Verzögerungen tatsächlich sind, hängt von der jeweiligen Implementierung und natürlich der Taktung der einzelnen Komponenten ab. Komplett vermeiden lässt sich der beim synchronen Anhalten entstehende Schlupf von einigen wenigen, typischerweise im einstelligen Bereich angesiedelten Befehlen jedoch nicht. Ob allerdings der jeweilige ARM-Controller diese Hardwareunterstützung bietet, obliegt dem Chip-Hersteller. Er ist frei in der Entscheidung, ob er die notwendigen CoreSight-Komponenten überhaupt auf dem Chip implementiert.

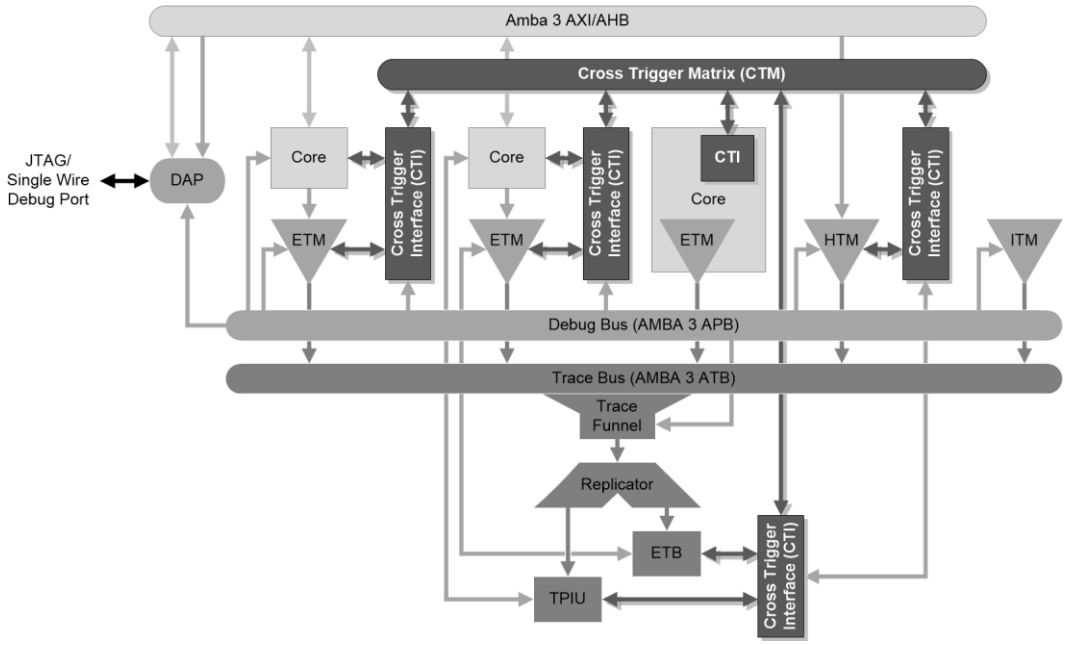

Abbildung 2: ARM CoreSight™ Debug- und Trace-Architektur mit Cross-Trigger-Einheiten

Auch von den Power-Architecture-basierten Controllern der MPC57xx-Familie wird das synchrone Run-Control hardwareseitig unterstützt. Die dafür verantwortliche Einheit heißt DCI (Debug and Calibration Interface). Der Vorteil gegenüber der ARM-Lösung: Wie beim Trigger-Switch des AURIX sind auch beim DCI die Peripherieeinheiten gleich mit angeschlossen, was ein Anhalten des gesamten Systems und nicht nur der Kerne erlaubt.

Freilich wäre es Entwicklern am liebsten, wenn sie sich mit solchen unterschiedlichen Gesichtspunkten gar nicht erst im Detail auseinandersetzten müssten. Debugger verbergen deshalb die unnötigen Details bezüglich der Konfiguration des synchronen Run-Controls hinter einer leicht zu bedienenden Oberfläche. Ein Beispiel dafür ist der Multi-Core-Run-Control-Manager in der Universal Debug Engine (UDE) der Firma PLS (**Abbildung 3**). Darin lassen sich Kerne zu Run-Control-Gruppen zusammenfassen, die dann beispielsweise an einem Breakpoint synchron angehalten und danach auch wieder synchron gestartet werden können.

Spuren mit Trace verfolgen

Gerade wenn es um Echtzeitanwendungen geht, gibt es neben dem synchronen Run-Control noch eine weitere wichtige Grundvoraussetzung für eine genaue und zuverlässige Systemanalyse: Den On-Chip-Trace. Natürlich steht diese Technologie auch bei den oben genannten Multicore-Controllern zur Verfügung. Freescale beispielsweise setzt für seine MPC57xx-Familie auf Nexus [2], Infineon nutzt dafür die Multi-Core-Debug Solution (MCDS) [3]und ARM bietet das bereits bekannte CoreSight. Allen gemein ist die Möglichkeit, Trace für mehrere Kerne parallel aufzuzeichnen, bei MCDS ist dies allerdings auf maximal zwei auswählbare Kerne limitiert. Zeitstempel erlauben die zeitliche Zuordnung der Trace-Daten, um die genaue Abfolge von Ereignissen zu rekonstruieren. Damit lassen sich zum einen Deadlocks und Race-Conditions aufspüren, zum anderen aber auch Flaschenhälse in der Kommunikation.

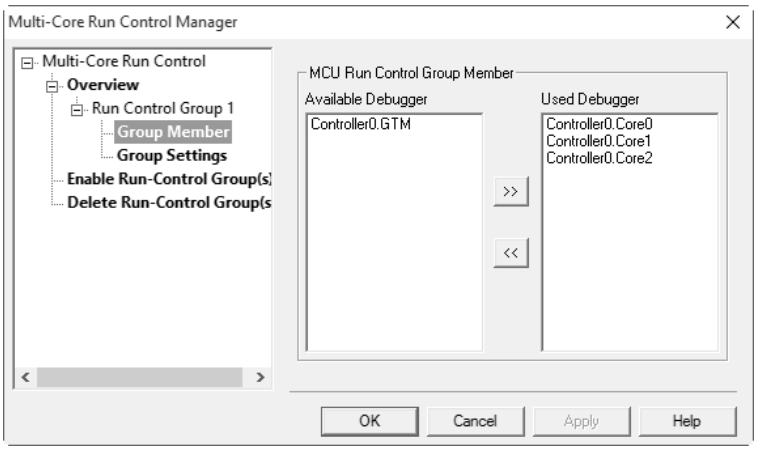

Abbildung 3: Multi-Core Run Control Manger der UDE zur Konfiguration des synchronen Haltens und Startens für einzelne Cores oder Core-Gruppen.

Eine große Herausforderung besteht nun darin, die aufgezeichneten Trace-Daten zum Debugger zu übertragen, der dann die weitere Analyse vornimmt. Entweder werden diese auf dem Chip in einem Trace-Buffer zwischengespeichert und per Debug-Schnittstelle ausgelesen, oder aber über ein breitbandiges Interface während der Aufzeichnung übertragen. Ersteres bietet natürlich eine viel höhere Bandbreite, aber nur eine sehr begrenzte Speicherkapazität. Letzteres erlaubt zwar eine theoretisch unbegrenzte Beobachtungsdauer, dafür kann es aber häufiger zu Überläufen kommen, wenn mehr Trace-Daten anfallen, als übertragen werden können. In beiden Fällen schaffen ausgeklügelte Filter- und Trigger-Mechanismen Abhilfe, welche die Menge der Trace-Daten einschränken. Auch sogenanntes Cross-Triggering ist damit möglich. Damit lässt sich beispielsweise der Trace für einen Kern starten, wenn eine Bedingung für einen anderen Kern wahr wird. Hilfreich ist diese Funktion beispielsweise, um Kommunikationen zwischen den Kernen zu debuggen. Bei MCDS und CoreSight zählt Cross-Triggering zu den Standardfunktionen. Allerdings konkurriert das es bei CoreSight mit dem synchronen Run-Control, da beide die gleichen Hardwareressourcen verwenden. Freescale hingegen musste seine Nexus-Implementierung extra um eine proprietäre Einheit, die sogenannte Sequence Processing Unit (SPU) erweitern, da im Nexus-Standard Cross-Triggering nicht vorgesehen ist.

Auch bei der zielgerichteten Beobachtung des Systemverhaltens mittels Trace sowie der anschließenden Auswertung und Analyse sind die Fähigkeiten des Debuggers gefragt. Für die Erstellung von Trace-Aufgaben stellt die UDE beispielsweise ein grafisches Werkzeug zur Verfügung, mit dem sich selbst komplexe Cross-Trigger recht einfach konfigurieren lassen. Das Ganze funktioniert für verschiedene On-Chip-Trace-Systeme, ohne dass sich der Anwender um die technischen Details kümmern muss. Verschiedene Ansichten, die beispielsweise die parallele Ausführung von Code auf mehreren Kernen visualisieren, erleichtern die Trace-Auswertung. Falls gewünscht oder benötigt, lassen sich mit Hilfe des Debuggers auch Profiling-Informationen gewinnen oder Code-Coverages bestimmen. Diese beiden Optionen sind jedoch nicht multicore-spezifisch, sondern vielmehr für die Systemanalyse und -optimierung von allgemeinem Nutzen.

Fazit

Ohne die Unterstützung durch geeignete Hardware auf dem Chip wäre das Multi-Core Debugging für tief eingebettete Systeme nur sehr mühsam zu bewerkstelligen. Auch ein moderner Debugger stößt unweigerlich an seine Grenzen, wenn es um synchrones Anhalten und Starten von mehreren Kernen geht. Wirklich synchrones Run-Control wird überhaupt erst durch geeignete On-Chip-Debug-Hardware mit konfigurierbaren Cross-Triggern möglich. Ähnliches gilt für die umfassende System-analyse von Multicore-Applikationen. Ohne On-Chip-Trace sind auch hier die Grenzen des Machbaren schon schnell erreicht. Zwar folgen die Chiphersteller keinem einheitlichen Standard, wenn es um On-Chip-Debugging geht, jedoch kommen moderne Debugger durchaus gut damit zurecht. Und moderne Debugger wie die UDE erlauben dem Entwickler eine einfache Nutzung der Funktionen, so dass er sich nur selten mit den Chip-spezifischen Besonderheiten beschäftigen muss.

Quellen

[1] ARM Ltd: *CoreSight Debug and Trace*; http://www.arm.com/products/system-ip/debug-trace/

[2] Nexus 5001 Forum: *IEEE-ISTO 5001™-2012, The Nexus 5001™ Forum Standard for a Global Embedded Processor Debug Interface*; http://nexus5001.org

[3] A. Mayer, H. Siebert, C. Lipsky: *Multi-Core Debug Solution IP*; White Paper, IPextreme, 2007

Autor

Jens Braunes ist Software-Architekt bei der PLS Programmierbare Logik & Systeme GmbH. Er ist dort im Softwareteam an der Weiterentwicklung der innovativen modularen Test- und Entwicklungstools von PLS beteiligt. Sein Fokus liegt dabei auf dem Design von Softwarekonzepten zur Anwendung von On-Chip-Emulatoren.

Jens Braunes ist regelmäßig als Autor von Fachartikeln und als Referent Fachtagungen tätig.

Kontakt

Internet: www.pls-mc.com
Email: jens.braunes@pls-mc.com

PLS Programmierbare Logik & Systeme GmbH
Technologiepark
02991 Lauta
Tel. +49 / 35722 -384 – 0