5. Übung	Fachgebiet Architektur eingebetteter Systeme **Hardwarepraktikum (TI)**	AES Embedded Systems Architecture SoSe 20

Ausgabe: 18.05.2020

Theorie Präsentation: 02.06.2020
Praxis Abgabe: 14.06.2020

Arbeitsziel

Implementierung von Komponenten ohne eigenen Code mittels Assistent und praktischer Funktionstest aller Komponenten außerhalb des Prozessorkerns auf dem FPGA. Grundverständnis für die Codierung von ARM-Instruktionen, Implementierung eines Befehlsgruppendecoders.

Hinweis: Ursprünglich wurde angekündigt, dass die Praxistests dieses Semester nicht stattfinden würden. Inzwischen haben wir jedoch Zugang zu den FPGA-Boards und können die Praxistests optional anbieten. Die entsprechenden Aufgabenteile (Aufgabe 3) sind grau hervorgehoben und können von Ihnen bei Interesse nachvollzogen werden. Wenn Sie die Möglichkeit zum Praxistest wahrnehmen wollen, senden Sie Ihr erzeugtes Bitfile vor der Rücksprache an `hardwprakt@aes.tu-berlin.de`.

Arbeitsmaterialien

- Modul `ArmUncoreTop.vhd`
- Modul `ArmChipSelectGenerator.vhd`
- Modul `ArmSwitchDebounce.vhd`
- Modul `ArmSystemController.vhd`
- Testbench `ArmUncoreTop_tb.vhd`
- EDIF-Netzliste `ArmMemInterface.edf`
- EDIF-Netzliste `ArmRS232Interface.edf`
- ARMv4-Befehlstabelle
- Modul `ArmCoarseInstructionDecoder.vhd`
- Tesbench `ArmCoarseInstructionDecoder_tb.vhd`
- Testvektordatei `COARSE_INSTRUCTION_DECODER_TESTDATA`

Abgabedateien

- Datei `ArmClkGen_sim_netlist.vhdl`
- Datei `ArmCoarseInstructionDecoder.vhd`
- `.vhd`-Dateien aus vorherigen Aufgaben (siehe *Hinweis zur Abgabe* in Aufgabe 2)
- Synthesereport von ArmUncoreTop (`ArmUncoreTop.vds`)
- Utilization Report von ArmUncoreTop (`ArmUncoreTop_utilization_synth.rpt`)

Theoretische Vorbetrachtungen

Die folgenden Fragen sind von der eingeteilten Gruppe in einem kurzen Vortrag zu beantworten. Alle Fragen bezüglich der ARM-Architekur beziehen sich immer auf ISA ARMv4:

- Was wird in der Digitaltechnik unter Taktversatz (clock skew) verstanden und warum kann selbiger Probleme verursachen?
- Welche Möglichkeiten bieten FPGAs zu Minimierung des Taktversatzes?
- Was versteht man im Zusammenhang mit FPGAs unter einem IP-Core? Was ist der Unterschied zwischen einem Soft- und einem Hard-IP-Core?

- Ein Teil von Aufgabenblatt 4 war die Realisierung des Transmitters einer asynchron-seriellen Schnittstelle, die lediglich über die beiden Leitungen `TXD` zum Senden und `RXD` zum Empfangen von Daten verfügte. Der vorgegebene Empfänger puffert jeweils nur ein Byte (der Inhalt des Registers `RS232_RCV_REG`), das jeweils nach Empfang der nächsten vollständigen Datenübertragung überschrieben wird. Werden die empfangenen Daten nicht schnell genug durch den Prozessor verarbeitet (z.B. über den Datenbus aus `RS232_RCV_REG` gelesen und in den Arbeitsspeicher kopiert), gehen sie verloren.
  Es gibt zwei verschiedene Ansätze, um den jeweils sendenden Kommunikationspartner einer RS232-Datenübertragung anzuhalten (*flow control*, *Datenflusssteuerung*, *handshake* oder *Synchronisation* genannt) und so Datenverluste zu verhindern. Ein Ansatz ist hardwarebasiert (zusätzliche Steuerleitungen), der andere softwarebasiert. Erklären Sie kurz die Bedeutung der optionalen Steuerleitungen *RTS* und *CTS* einer seriellen Schnittstelle und die Funktionsweise des *XON/XOFF*-Protokolls.

Empfohlene Quellen zur Bearbeitung:

- Mikroprozessortechnik und Rechnerstrukturen, [1, Kapitel 7.5]
- Moderne Prozessorarchitekturen, [2, Kapitel 2.2.3]

Aufgabenbeschreibung

In dieser Aufgabe werden die außerhalb des Prozessorkerns liegenden Komponenten vervollständigt, miteinander verbunden und schließlich praktisch getestet. Zudem wird ein Befehlsgruppenencoder implementiert.

Aufgabe 1 (2 Punkte) Erzeugung eines Kerntaktes mit und ohne Phasenverschiebung

Das FPGA auf dem Basys wird mit einem externen 100-MHz-Taktsignal aus einem Taktgeber versorgt. Um diesen Takt unmittelbar für den Prozessor verwenden zu können, müsste die Signallaufzeit über den kritischen Pfad des Prozessors und der Peripherie unter 10 ns liegen. Dies ist nicht der Fall. Darüber hinaus wird für Arbeitsspeicher und Registerspeicher ein gegenüber allen anderen synchronen Komponenten `180°` phasenverschobener Takt benötigt.

Für die Synthese neuer Taktsignale sind im FPGA sogenannte CMTs (*Clock Management Tiles*) vorgesehen. Sie bestehen aus einem MMCM (mixed-mode clock manager) und einer PLL (phase-locked loop). Sie erzeugen auf Basis eines Eingangstaktes einen neuen Takt mit höherer oder niedrigerer Frequenz und bei Bedarf auch einen Phasenversatz zwischen Eingangs- und Ausgangstakt.

Zur unmittelbaren Verwendung eines dieser beider Komponenten in VHDL-Modulen müssen spezielle Vorlagen (*Language Templates*) verwendet werden. Sie sind in Vivado, neben vielen anderen, im Menü unter *Tools* → *Language Templates* zugänglich (oder auch in der Flow-Navigator Seitenleiste mit dem gleichen Namen unter der *Project Manager* Gruppe). Hier soll aber der Weg über einen Assistenten (*Clocking Wizard*) zur geführten Erzeugung von Komponenten beschritten werden. Insbesondere müssen die dabei erzeugten Templates nicht mehr von Hand an die eigenen Bedürfnisse angepasst werden.

Dafür wird der *IP Catalog*, der in der *Project Manager* Gruppe steht, genutzt. Nachdem Sie darauf gedrückt haben, öffnet sich dieser. Am einfachsten ist es mit dem *Search-Toolbar* den *Clocking Wizard* zu suchen und ihn dann auszuwählen.

Sie werden durch folgende Dialoge geführt und können die Einstellungen mit Abb.1 vergleichen:

- Um nicht von den ganzen möglichen Ports überflutet zu werden, kann man oben links das Häkchen bei *Show disabled Ports* entfernen.
- Geben Sie zunächst oben in der Leiste *Component Name* `ArmClkGen` als Name ein.
- Sie befinden sich im Tab *Clocking Options*. Achten Sie auch darauf, dass die *Input Frequenqy(MHz)* gleich `100.000 MHz` ist.
- Öffnen Sie nun den Reiter *Output Clocks*. Setzen Sie dort einen weiteren Haken bei `clk_out2`. Ändern Sie **beide** *Output Freq (MHz)* auf `10 MHz`. Setzen Sie zudem die Phasenverschiebung (*Phase (degrees)*) von `clk_out2` auf 180°.
- Vergleichen Sie die Einstellungen mit Abb.1
- Bestätigen Sie ihre Einstellungen.
- Danach öffnet sich ein Fenster *Generate Output Product*. Klicken Sie hier einfach auf *Generate*.

In den Projektquellen (Fenster *IP Sources*) taucht nun die Komponente **ArmClkGen** auf, gekennzeichnet durch das IP-Icon. Ein Doppelklick auf die Komponente öffnet erneut den *Clocking Wizard*, wo sämtliche Einstellungen nachträglich geändert werden können. Die Datei, die nun abgegeben werden soll, befindet sich im Ordner:
`<Projekt>/<Projekt>.srcs/sources_1/ip/ArmClkGen/ArmClkGen_sim_netlist.vhdl`

Wenn Sie sich den Inhalt dieser Datei ansehen, können Sie sehen, dass Komponenten der *UNISIM-library* instanziiert worden sind. **Diese Datei sollten Sie auch zur Simulation unter QuestaSIM verwenden.**

Der eigentliche Wrapper, den Vivado als *Top-Modul* der IP nutzt heißt `ArmClkGen.vho`. (Diese soll **nicht** abgegeben werden).

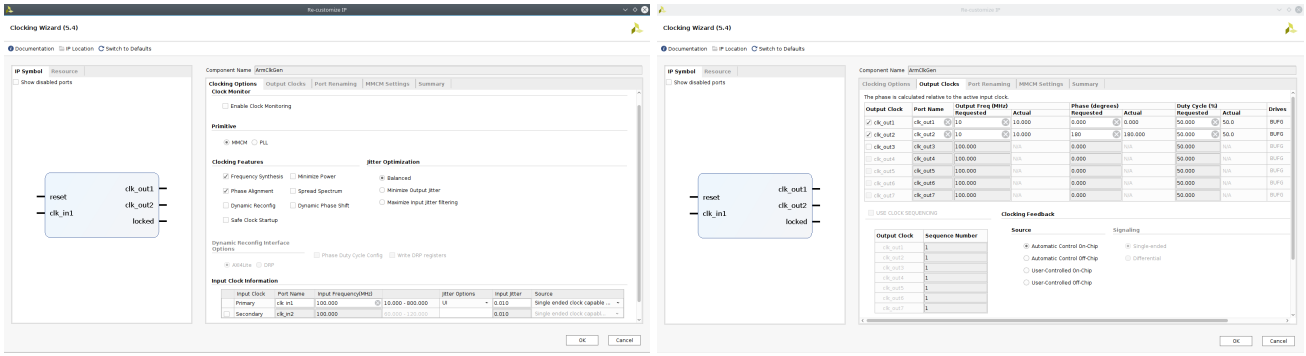

Abbildung 1: Clocking Wizard Einstellungen

Erklärung der getroffenen Einstellungen:
Die CMTs können Taktsignale entweder per MMCM (*mixed-mode clock manager*) oder per *phase-locked loop* (PLL) erzeugen. Die PLL ist weniger variabel in Ein- und Ausgangsfrequenz und kann als *vereinfachter* MMCM betrachtet werden. Der *locked*-Ausgang ist ein Kontrollsignal, an dem abgelesen werden kann, ob der neue Takt stabil ist, was einige tausend Perioden des Eingangstaktes dauern kann. Das Locksignal wird später verwendet um System Controller und Prozessor zu starten, sobald der Kerntakt stabil vorliegt.

Eine Instanz des von Ihnen generierten Taktgebers wird bereits in der Vorgabe **ArmSystemController.vhd** (den Sie spätestens jetzt ins Projekt importieren) verwendet. Natürlich kann der Taktgeber an einer beliebigen Stelle und Hierarchieebene im Design eingebunden werden. Da der System-Controller allerdings auf die Stabilisierung des Kerntakts warten muss, bevor er den Arbeitsspeicher mit einem Programm initialisieren kann, ist es sinnvoll, den Taktgenerator in diesem Modul zu instanziieren.

Allerdings ist der System-Controller für das Laden von Anwendung und einen anschließenden Neustart des Prozessors verantwortlich. Er wartet dazu die Stabilisierung des Kerntaktes ab (LOCK-Signal des CMT), nutzt die ebenfalls mit Kerntakt operierende serielle Schnittstelle zum Laden eines Programms und deaktiviert anschließend das Resetsignal des Prozessors, der damit letztlich gestartet wird. Insofern ist es passend, die Takterzeugung im System-Controller anzusiedeln, wo das LOCK-Signal ausgewertet werden muss.

Aufgabe 2 (3 Punkte) Verhaltenssimulation aller Komponenten außerhalb des Prozessorkerns

Kompilieren Sie alle Vorgaben, insbesondere das Modul **ArmUncoreTop.vhd** und die zugehörige Testbench **ArmUncoreTop_TB.vhd** in QuestaSim. **ArmUncoreTop** enthält alle Komponenten unseres kleinen Prozessorsystems, die nicht unmittelbar Teil des Prozessorkerns sind: Der Datenbus ist in Form einiger Signale realisiert und mit dem Speicherinterface **ArmMemInterface** verbunden (der Instruktionsbus wird ohne Prozessorkern nicht benötigt). Ebenfalls an den Datenbus angebunden ist die von Ihnen teilweise implementierte serielle Schnittstelle **ArmRS232Interface**. Hinzu kommt das Modul **ArmSystemController.vhd**.

Hinweis zur Abgabe In dieser Aufgabe verwenden Sie mehrere Komponenten aus vorherigen Aufgaben. Bitte geben Sie **alle** `.vhd`-Dateien der Komponenten ab, die Sie in **ArmUncoreTop.vhd** benutzen. Sollten Sie keine Simulationsmodelle nutzen (siehe Hinweis unten), sind das folgende Dateien:

- `ArmMemInterface.vhd`
- `ArmRAMB_4kx32.vhd`
- `ArmRAMB_4kx8.vhd`
- `ArmRS232Interface.vhd`
- `ArmPISOShiftReg.vhd` (falls in ArmRS232Interface verwendet)
- `ArmWaitStateGenAsync.vhd` (falls in ArmRS232Interface verwendet)

Funktionsweise des von ArmUncoreTop Sobald der Taktgeber ein stabiles Taktsignal durch *LOCKED* anzeigt, beginnt der System-Controller, den Speicher zu initialisieren. Dazu liest er über den Datenbus permanent das Statusregister der seriellen Schnittstelle aus, liest ggf. ein vorhandenes Datum aus dem Empfangsregister und kopiert es in den Arbeitsspeicher, beginnend an Adresse 0. Dies wiederholt sich, bis die durch `ARM_PROG_MEM_SIZE` aus dem Paket **ArmConfiguration** angezeigte Menge von Bytes eingelesen und in den Speicher kopiert wurde.

Anschließend liest der System-Controller den Speicher aus und schreibt dessen Inhalt zur Kontrolle wieder vollständig in die serielle Schnittstelle. Danach gibt er den Datenbus frei und führt ggf. ein (internes) Reset des Prozessors durch, durch das der Inhalt des Arbeitsspeichers aber gelöscht wird.
Der System-Controller initialisiert den Speicher nach Auftreten eines externen Resetsignals (ein Taster auf dem Board) genau einmal, wobei er das niederwertigste Byte der aktuellen Speicheradresse auf einem Statusausgang anzeigt. Der Statusausgang wird im Topmodul aus dem Prozessorsystem herausgeführt und mit den 8 LEDs auf dem Starter Kit verbunden. Nach der Speicherinitialisierung zeigt der Statusausgang des fertigen Prozessorsystems die Bits (9:2) der aktuellen Instruktionsbusadresse (also die 8 niederwertigen Bits der Instruktions-Wortadresse).

Ablauf des Tests Simulieren Sie mithilfe der Testbench **ArmUncoreTop_TB**, ob das Initialisieren des Speichers wie gewünscht durchgeführt wird. Die Testbench legt ein simuliertes 100-Mhz-Taktsignal an, führt ein Reset durch und simuliert die Übertragung der Speicher-Initialwerte über die serielle Schnittstelle mit Zufallswerten. Anschließend liest die Testbench die vom System-Controller zurückgelieferten Kontrolldaten ein und vergleicht sie mit den gesendeten. Erfolgt keine Antwort oder werden zu wenige Daten zurückgeschickt, tritt ein Timeout ein. Die gesamte Prozedur wird, inkl. des Resets, zweimal durchgeführt.

Die Testbench überprüft lediglich die grundlegende Funktionalität von **ArmUncoreTop**, Spezialtests, beispielsweise bzgl. des Timings der seriellen Schnittstelle, sind nicht enthalten. Sollten Fehler auftreten, suchen Sie deren Ursache selbstständig in QuestaSim. Berücksichtigen Sie dabei, dass nicht nur der Verlauf der Schnittstellensignale von **ArmUncoreTop**, sondern der Verlauf aller Signale, Variablen und selbstdefinierter Speicher (Arrays) aller instanziierten Module in QuestaSim angezeigt werden kann.

HINWEIS

Setzen Sie für eine erste Simulation `ARM_PROG_MEM_SIZE` auf einen niedrigen Wert, z.B. 128. Dieser Wert wird über die `ArmConfiguration.vhd`-Datei geändert. Auf diese Weise halten sich Simulationszeit und anfallende Datenmenge für die Fehlersuche in Grenzen.

Es wird für eine komplette Simulation circa 15 min dauern. Normalerweise sollten Sie nun ein Test mit 16384 Bytes durchführen, da es aber ein Update in der UNISIM Bibliothek gab, würde die Simulation mit dieser Größe, viele Stunden dauern. Deswegen reicht ein Durchlauf mit 128 Byte zwar für die Aufgabe aus, aber es wird nicht die Fehlerfreiheit des Systems garantieren. Stellen sie aber unbedingt `ARM_PROG_MEM_SIZE` für die Synthese auf den Wert 16384 Byte zurück.
Sind in einer der bisherigen Testbenches zu den in **ArmUncoreTop** instanziierten Modulen unkorrigierte Fehler aufgetreten, würden sich diese nun erneut bemerkbar machen. Damit das nicht geschieht und Sie nur mit fehlerfrei implementierten Modulen arbeiten, werden Ihnen zukünftig vorkompilierte Simulationsmodelle zur Verfügung gestellt. Diese Modelle bilden die in QuestaSim eingebundene Bibliothek **ARM_SIM_LIB**[1].

Falls ihre Implementierung bisher nicht fehlerfrei war, können Sie für dieses Aufgabenblatt die Simulationsmodelle **ARM_SIM_LIB.ArmRS232Interface** oder **ARM_SIM_LIB.ArmMemInterface** nutzen. Um die Modelle zu verwenden, kommentieren Sie die entsprechenden Zeilen zur Bibliothek **ARM_SIM_LIB** am Beginn von **ArmUncoreTop.vhd** ein und die entsprechenden Zeilen zur Bibliothek *work* aus:

```
library ARM_SIM_LIB;
    use ARM_SIM_LIB.ArmRS232Interface;
--  use ARM_SIM_LIB.ArmMemInterface;
library work;
--  use work.ArmRS232Interface;
    use work.ArmMemInterface;
```

Im Beispiel wird das vorgegebene Simulationsmodell der seriellen Schnittstelle, jedoch Ihre Implementierung des Arbeitsspeichers verwendet. Für die Synthese ist es später notwendig, alle USE-Anweisungen zur Simulationsbibliothek und die Zeile

```
    LIBRARY ARM_SIM_LIB;
```

auszukommentieren, da Vivado die Simulationsbibliothek nicht verarbeiten kann. Die Verwendung der Simulationsmodelle ist nur dann gestattet, wenn bereits Punkte auf die fehlerhaft implementierten Module abgezogen wurden.

---

[1] Die Bibliothek ist automatisch in QuestaSim eingebunden, wenn sie direkt auf dem Server arbeiten. Falls dies bei Ihnen nicht der Fall ist, wählen Sie bei QuestaSim *File* → *New* → *Library*. Wählen Sie dann unbedingt *a map to an existing library*. Der Name der Bibliothek ist `ARM_SIM_LIB` und sie befindet sich unter:
/afs/tu-berlin.de/units/Fak_IV/aes/lib/hwpti/ARM_SIM_LIB

Aufgabe 3 (2 Punkte) Synthese und Praxistest

Synthese Erklären Sie **ArmUncoreTop** in Vivado zum Topmodul. Sollten Sie, aufgrund von eigenen, fehlerhaften Implementierungen in älteren Aufgabenblättern, vorgegebene Simulationsmodelle genutzt haben, so müssen die für die Synthese in dem Modul **ArmUncoreTop.vhd** auskommentiert werden. Benutzen Sie stattdessen die von uns zur Verfügung gestellten Vorgaben (siehe Hinweis)

Synthetisieren Sie ihr Design und überprüfen Sie den Synthesereport. Er wird einige nicht zu vermeidende Warnings, auch in vorgegebenen Modulen, enthalten. Darunter sollten jedoch keine bzgl. des Vorhandenseins von Latches oder asynchronen Rückkopplungen (combinatorial loops) sein.

Geben Sie den Synthese-Report (`ArmUncoreTop.vds`) und den Utilization-Report (`ArmUncoreTop_utilization_synth.rpt`) ab. Sie finden beide Dateien im Projektordner unter `Projektname.runs/synth_*/`.

Praxistest Erzeugen Sie eine XDC-Datei für die Signale des Topmoduls mit folgender Abbildung von Portsignalen auf Leitungen und Bauteile des Boards:

`EXT_CLK`	Verbunden mit dem 100-MHz-Taktgeber.
`EXT_RST`	Verbunden mit dem „unteren“ Button.
`EXT_LDP`	Verbunden mit Switch 0.
`EXT_RXD`	Verbunden mit der RXD-Leitung des USB-RS232 Interfaces.
`EXT_TXD`	Verbunden mit der TXD-Leitung des USB-RS232 Interfaces.
`EXT_LED`	Verbunden mit den acht unteren LEDs des Boards.

Erzeugen Sie ein Bitfile Ihres Designs. Konfigurieren Sie anschließend das FPGA und lassen Sie den Betreuer die Implementierung testen. Dabei wird der Speicher über die serielle Schnittstelle von einem Entwicklungsrechner mit Zufallsdaten aus einer Datei initialisiert und die Antwort des System-Controllers in eine weitere Datei geschrieben. Ein Vergleich beider Dateien darf keine Unterschiede zutage fördern.

HINWEIS

Die Synthese von Modulen, die in den bisherigen Tests Fehler gezeigt haben, ist nicht sinnvoll. Daher stellen wir auch hier fehlerfreie Vorgaben zur Verfügung. Verwenden Sie die Vorgaben nur, wenn bereits Punkte auf fehlerhaft implementierte Aufgaben abgezogen wurden. Die Komponenten werden als sogenannte EDIF-Netzlisten bereitgestellt. Sie sind das Ergebnis der Synthese von Modulen in *Vivado* und noch architekturunabhängig. Die Vorgaben liegen im AFS im Verzeichnis */afs/tu-berlin.de/units/Fak_IV/aes/lib/hwpti/ARM_LIB*. Sie werden nicht in ihr Projektverzeichnis kopiert, sondern von dort aus eingebunden.

Gehen Sie zur Verwendung der Vorgaben folgendermaßen vor:

- Entfernen Sie alle bisher fehlerhaften Quellen aus dem Projekt (rechte Maustaste auf die Komponente → *Remove File from Project*). Die Dateien werden dabei zwar nicht aus dem Quellenverzeichnis gelöscht, ein Fragezeichen vor der Komponente zeigt aber an, das entsprechende Instanzen keiner Quelle mehr zugeordnet sind
- Fügen Sie die EDIF-Dateien über den *Add Sources* Dialog hinzu, genau so wie Sie es von VHDL-Dateien gewohnt sind. Anstelle des runden *vh*-Icons erscheint nun ein kleines Icon mit einem *N* neben der Komponente im *Sources*-Fenster.

Aufgabe 4 (5 Punkte) Implementierung eines Instruktionsgruppendecoders

Nachdem Sie nun die Komponenten außerhalb des Prozessorkerns implementiert und ausgiebig getestet haben, beginnen Sie mit der Implementierung von Komponenten innerhalb des Kerns.

Jeder aus dem Speicher gelesene Instruktionsvektor wird am Beginn der ID-Stufe des Kontrollpfades einer von 17 Instruktionsgruppen grob zugeordnet. Die Gruppen sind nicht durch die ISA definiert, sondern willkürlich so gewählt, dass die Instruktionen innerhalb einer Gruppe sehr ähnliche Befehlsvektoren aufweisen. Die Zuordnung erfolgt durch den Instruktionsgruppendecoder **ArmCoarseInstructionDecoder.vhd**. Diese Maßnahme vereinfacht anschließend die Implementierung der Prozessor-Hauptsteuerung, da dort nur noch wenig zusätzliche Decodierarbeit zu leisten ist. In der Hauptsteuerung wird erkannt, um welche Instruktion innerhalb der bereits bestimmten Gruppe es sich handelt und anschließend werden die notwendigen Steuersignale zur ihrer Ausführung erzeugt.

Vervollständigen Sie das Modul **ArmCoarseInstructionDecoder.vhd**. Dieser Instruktionsgruppendecoder, dessen Schnittstelle der Tabelle 1 entnommen werden kann, nimmt ein vollständiges Instruktionswort entgegen und liefert die zugehörige Instruktionsgruppe *one-hot*-codiert zurück.

`CID_INSTRUCTION(31:0)`	`in`	Eine aus dem Instruktionsspeicher gelesene Instruktion.
`CID_DECODED_VECTOR(15:0)`	`out`	Codierung der erkannten Instruktionsgruppe gemäß der Konstanten des Typs `COARSE_DECODE_TYPE` aus **ArmTypes.vhd**

Tabelle 1: Schnittstelle des Instruktionsgruppendecoders

Im ARM-Instruktionssatz werden die Befehlsbits (27:20) und (7:4) eines Befehlswortes zur Unterscheidung der Instruktion verwendet. Unter Auswertung dieser 12 Bits kann jede Instruktion also eindeutig identifiziert werden. Zum einen soll aber in dieser Aufgabe keine eindeutige Trennung nach Instruktionen durchgeführt werden (ADD, SUB, MOV...) sondern nach Instruktionsgruppen. Zum anderen wäre die Verwendung eines 12-Bit-Decoders offensichtlich realitätsfern (realisiert als ROM: 4096 Zeilen). Stattdessen sollen jeweils nur die Bits eines Befehlswortes zur Befehlsgruppenerkennung herangezogen werden, die unbedingt notwendig sind. Beispielsweise ist die SWI-Befehlsgruppe eindeutig daran zu erkennen, dass die Bits 27:24 des Instruktionswortes den Wert `1111` haben.

Die Zuordnung von Instruktionen zu Gruppen können den beiden letzten Spalten der ARMv4-Befehlstabelle (Teil der Befehlssatzarchitektur-Dokumentation im Begleitmaterial des Hardwarepraktikums) entnommen werden. Es gibt 16 reguläre Gruppen und zusätzlich *CD_UNDEFINED*. Einige Gruppen umfassen nur eine einzige Instruktion, maximal gehören 16 Befehle zu einer Gruppe. Ein Befehl kann auch zu mehreren Gruppen gehören. Im Wesentlichen erfolgt die Gruppenzuordnung nach folgendem Muster:

- Alle Coprozessorbefehle werden einer Gruppe zugeordnet (wir implementieren keine Coprozessoren, bereiten die Prozessorsteuerung aber darauf vor).
- Wo Operand-2 (Bits 11:0 vieler Instruktionen) in unterschiedlichen Formen angegeben werden kann (Load-Store-Befehle, Arithmetisch-logische Befehle, MSR-Befehl), entspricht jede Form einer eigenen Befehlsgruppe. Ist Operand-2 ein Direktoperand, hat der Gruppenname das Suffix `IMMEDIATE` (z.B. `CD_ARITH_IMMEDIATE`). Ist Operand-2 ein Register, dessen Inhalt aber anhand einer direkt angegebenen Schiebeweite manipuliert wird, so lautet das Suffix

REGISTER. Bei Arithmetischen Operationen kann auch die Schiebeweite ein Registerinhalt sein, die entsprechende Konstante lautet CD_ARITH_REGISTER_REGISTER.

- Befehlsworte, die nicht zu einer der definierten Instruktionen gehören, werden durch den Decoder der Gruppe CD_UNDEFINED zugeordnet.
- SBZ- und SBO-Felder werden durch den Decoder nicht ausgewertet.
- Für die Identifikation der Multiplikations-Befehlsgruppe sind alle 8 möglichen Kombinationen des Mul-Feldes in Multiplikationsbefehlen erlaubt. Die geeignete Reaktion auf nicht implementierte Multiplikationsbefehle erfolgt anderweitig in der Prozessorsteuerung.
- Die in der Befehlstabelle konkret vorgegebenen Bits (1/0) sind in den jeweiligen Instruktionen obligatorisch, selbst wenn sie nicht zur Abgrenzung von anderen Befehlen benötigt werden. Ein STR-Befehl mit Registeroperand enthält an Bitposition 4 immer eine 0, obwohl er anderweitig eindeutig identifiziert werden kann. Ein Verstoß gegen die obligatorischen Werte wird als unbekannte Instruktion (CD_UNDEFINED) behandelt.
- Die gelb gefärbten Felder enthalten beliebige Direktoperanden oder Adressen von Registeroperanden und werden vom Gruppendecoder nicht ausgewertet.
- Die in der Befehlstabelle blau hinterlegten Felder differenzieren normalerweise die Instruktionen einer Gruppe untereinander oder bestimmen die Detailwirkung einer Instruktion, z.B. die Adressierungsart. Der Gruppendecoder kann diese Felder weitgehend ignorieren. Die Ausnahmen von der Regel sind:
  - Bit 22 unterscheidet die Gruppen CD_LOAD_STORE_SIGNED_IMMEDIATE und -_REGISTER.
  - Die Instruktionen LDRSB, LDRH, LDRSH, STRH enthalten niemals die Kombination 00 für S- und H-Bit. Stattdessen handelt es sich dann um sehr ähnliche Befehlsvektoren der Multiplikations- oder Swap-Instruktionen oder um einen Vektor, dem keine Instruktion zugeordnet werden kann.
  - Ebenfalls weisen die Bits L und S dieser vier Instruktionen niemals die Kombination L = 0, S = 1 auf (damit würde eine sinnlose „Store Signed“ Instruktion angezeigt). Entsprechende Befehlsvektoren müssen CD_UNDEFINED zugeordnet werden.
  - Die 16 Arithmetisch-logischen Instruktionen sind durch ihren vierstelligen Opcode differenziert. Für die vier Opcodes 10--, muss das S-Bit aber zwingend gesetzt sein. Andernfalls handelt es sich, in Abhängigkeit von Instruktionsbit 21, 25 und ggf. auch 4 um eine MSR-Instruktion, MRS-Instruktion oder einen undefinierten Befehlsvektor.

Die Unterscheidung der Gruppen ist zur besseren Übersichtlichkeit in einem Entscheidungsdiagramm dargestellt, das Sie in Abbildung 2 finden. INST(high:low) bezeichnet dabei einen Abschnitt des Instruktionsvektors. Setzen Sie das Entscheidungsdiagramm in VHDL um. Eine Formulierung innerhalb eines kombinatorischen Prozesses mit case- und if-then-else-
Anweisungen bietet sich an. Weisen Sie dem Ausgang CID_DECODED_VECTOR ausschließlich die vorgegebenen Konstanten des Typs COARSE_DECODE_TYPE aus dem Package ArmTypes zu. Verwenden Sie keine explizite Signalcodierung. Auf diese Weise ist es möglich, die Codierung der Instruktionsgruppen nachträglich zu bearbeiten ohne den Decoder ändern zu müssen.

Kopieren Sie die Datei COARSE_INSTRUCTION_DECODER_TESTDATA in Ihr Testvektorverzeichnis und führen Sie eine *Verhaltenssimulation* mit der Testbench **ArmCoarseInstructionDecoder_TB.vhd** durch.

HINWEIS

Der Auswahl-Ausdruck innerhalb einer VHDL-case-Anweisung (`case <Ausdruck> is`) darf ein zusammenhängender Ausschnitt eines Arrays/Vektors sein, jedoch keine Konkatenation verschiedener Signale. Sie können in diesem Fall ein zusätzliches Signal definieren, diesem eine Konkatenation von Vektorausschnitten zuweisen und das Signal dann als geschlossenen Auswahlausdruck in der `case`-Anweisung verwenden.
Am Ende des vorgegebenen Moduls befindet sich ein Abschnitt für die Simulation, in dem sichergestellt wird (Zusicherung einer Bedingung durch die `assert`-Anweisung), dass maximal ein Bit von `CID_DECODED_VECTOR` gesetzt ist. Verwenden Sie unbedingt das vorgegebene Signal `DECV`, dem Sie die Gruppe der aktuellen Instruktion zuweisen, damit dieser automatische Test keine Fehler in der Verhaltenssimulation erzeugt.

Mündliche Rücksprache - 4 Punkte

Literatur

[1] Thomas Flik. *Mikroprozessortechnik und Rechnerstrukturen*. 7., neu bearb. Aufl. Springer Berlin, 2004. ISBN: 3-540-22270-7.

[2] Matthias Menge. *Moderne Prozessorarchitekturen. Prinzipien und ihre Realisierungen*. 1. Aufl. Springer, Berlin, März 2005. ISBN: 3540243909.

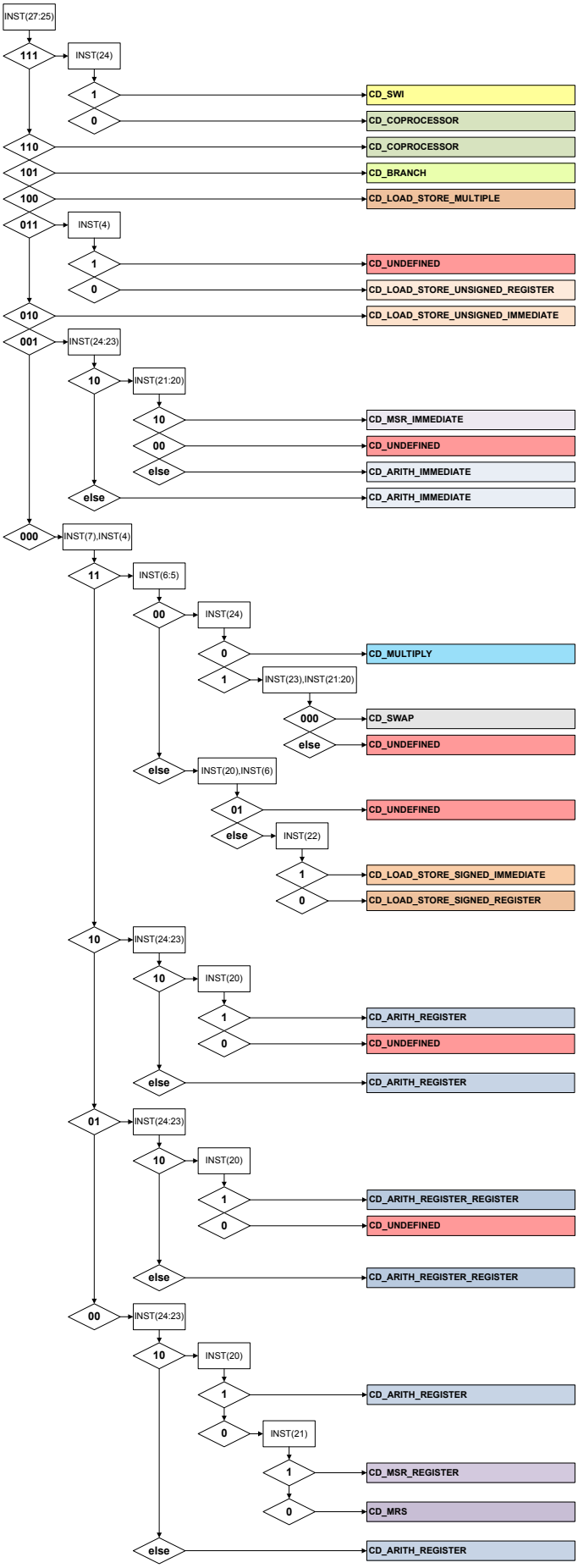

Abbildung 2: Entscheidungsdiagramm für den Instruktionsgruppendecoder