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

Ausgabe: 22.06.2020

Theorie Präsentation: 06./07.07.2020
Praxis Abgabe: 19.07.2020

Arbeitsziel

Aufgabenblatt 10 bildet den Abschluss des Hardwarepraktikums. Sie implementieren einen Teil der Prozessor-Hauptsteuerung. Anschließend wird die Funktion des gesamten Prozessorsystems anhand einer einfachen Beispielanwendung simuliert, dann das Gesamtsystem synthetisiert und in das FPGA übertragen und die gleiche Beispielanwendung dort ausgeführt.

Hinweis: Wie bei Blatt 5 ist der Praxistest dieses Semester optional. Wenn Sie die Möglichkeit zum Praxistest wahrnehmen wollen, senden Sie Ihr erzeugtes Bitfile vor der Rücksprache an `hardwprakt@aes.tu-berlin.de`.

Arbeitsmaterialien

- Dokumentation Prozessorsteuerung
- Modul `ArmTop.vhd`
- Modul `ArmCore.vhd`
- Modul `ArmControlPath.vhd`
- Modul `ArmConditionCheck.vhd`
- Modul `ArmArithInstructionCtrl.vhd`
- Modul `ArmDelayShiftRegister.vhd`
- Modul `ArmShiftRecoder.vhd`
- Modul `ArmCopDecoder.vhd`
- Testbench `ArmCore_tb.vhd`
- Testbench `ArmTop_tb.vhd`
- do-File `Mydo.udo`
- do-File `ArmTop_tb.udo`
- XDC `ArmTop.xdc`
- Beispielanwendung `arm_TEST_MAIN.c`
- Simulationsmodelle und Netzlisten bereits implementierter Module

Abgabedateien

- Datei `ArmArithInstructionCtrl.vhd`
- `.vhd`-Dateien bereits implementierter Module aus vorherigen Aufgaben
- Synthesereport von ArmTop (`ArmTop.vds`)
- Utilization Report von ArmTop (`ArmTop_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:

- Zur Programmierung des Hardwarepraktikums-Prozessors wird in Aufgabe 2 statt einer maschinennahen Programmierung in ARM-Assembler die Hochsprache ANSI C verwendet. Erläutern Sie in diesem Zusammenhang kurz die Begiffe *Compiler*, *Assembler* und *Linker*.
- Übersetzte Programme liegen in der Regel in einem bestimmten Binärformat (engl. *object format*) wie z.B. ELF oder COFF vor. Wieso können Sie nicht einfach das von Compiler und Linker erzeugte Binärprogramm direkt auf den Prozessor laden und ausführen?
- Betrachten Sie die Testanwendung ARM_TEST_MAIN.c aus den Vorgaben zum Aufgabenblatt, die über die serielle Schnittstelle die Zeichenkette *Hallo Welt* ausgibt. Wieso können Sie nicht einfach die Funktion *printf* wie für alle anderen geläufigen Hallo-Welt-Programme verwenden?

Aufgabenbeschreibung

Im Folgenden wird das gesamte Prozessorsystem mit vervollständigter Steuerung einer Verhaltenssimulation und einem Praxistest unterzogen.

Aufgabe 1 (6 Punkte) Prozessorsteuerung für arithmetisch-logische Instruktionen

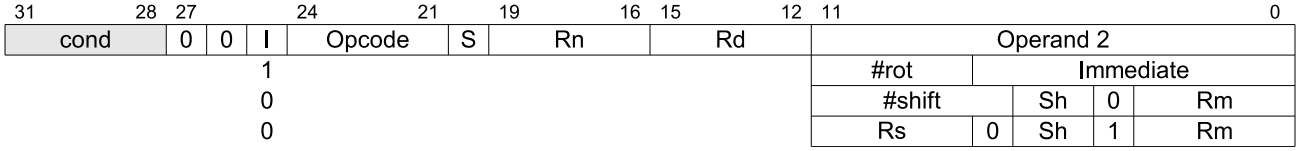
Die Prozessorsteuerung in **ArmControlPath.vhd** ist als Zweiprozess-Statemachine implementiert. Die möglichen Prozessorzustände entsprechen dem Typ `ARM_STATE_TYPE` im Package **ArmTypes.vhd**. Die genaue Bedeutung jedes einzelnen Zustandes ist der Dokumentation der Prozessorsteuerung zu entnehmen. Zum Verständnis der Aufgabe sind nur zwei Zustände von Bedeutung:
Im Zustand `STATE_DECODE` erzeugt die Prozessorsteuerung alle notwendigen Steuersignale, die zur Bearbeitung der aktuell im Instruktionsregister stehenden Instruktion benötigt werden. Ein Teil dieser Steuersignale wirkt unmittelbar in der *Decode*-Stufe auf den Datenpfad ein, beispielsweise die Leseadressen für Operanden aus dem Registerspeicher. Alle übrigen Steuersignale werden in Fließbandregister des Kontrollpfades geschrieben und wirken auf die Stufen *Execute*, *Memory* oder *Write-Back*, sobald die decodierte Instruktion diese Stufen durchläuft.
Verursacht eine Instruktion einen Sprung, muss die Prozessorsteuerung einige Takte warten, bis die Sprungzieladresse im Instruktionsadressregister erscheint. Erst dann kann die nächste Instruktion von dieser Adresse gelesen werden. Die Wartezeit überbrückt die Steuerung im Zustand `STATE_WAIT_TO_FETCH`, in dem alle Steuersignale so gesetzt werden, dass sie den Prozessorzustand nicht ändern.

Die Prozessorsteuerung, mit der Schnittstelle aus Tabelle 1, erzeugt Steuersignale in Abhängigkeit von der im Gruppendecoder bestimmten Instruktionsgruppe. Für die Gruppen `CD_ARITH_IMMEDIATE`, `CD_ARITH_REGISTER` und `CD_ARITH_REGISTER_REGISTER` wurde die Erzeugung der Steuersignale in das zusätzliche Modul **ArmArithInstructionCtrl.vhd** ausgelagert, das von Ihnen nun vervollständigt werden soll.
Die Namen der meisten Ausgänge des Moduls sind an den Steuereingängen des Datenpfades orientiert. Steuereingänge des Datenpfades, die keine Entsprechung in **ArmArithInstructionCtrl.vhd** haben, sind für arithmetisch-logische Instruktionen nicht von Bedeutung und/oder wurden an anderer Stelle in der Prozessorsteuerung bereits mit einem Initialwert belegt, gewöhnlich `0`. Betroffen sind hiervon beispielsweise die diversen 2:1-Multiplexer-Steuerleitungen der *Execute*-Stufe.

AIC_DECODED_VECTOR(15:0)	Vom Instruktionsgruppendecoder ermittelte Gruppe der aktuellen Instruktion gemäß des Typs COARSE_DECODE_TYPE.
AIC_INSTRUCTION(31:0)	Befehlswort der aktuellen Instruktion.
AIC_IF_IAR_INC	Inkrement-Steuersignal für das Instruktionsadressregister im Datenpfad.
AIC_ID_R_PORT_(X)_ADDR(3:0)	(X = A, B, C) Registeradressen von bis zu 3 Operanden. Verwendet werden hier noch die 4-Bit-Adressen der ARM-Architektur.
AIC_ID_REGS_USED(2:0)	Steuersignale um anzuzeigen, welche Operanden für die Instruktion aus dem Registerspeicher gelesen werden müssen, entspricht ABC_INST_REGS_USED der Bypasssteuerung.
AIC_IMMEDIATE(31:0)	Ein für die Befehlsausführung ggf. benötigter Direktoperand.
AIC_ID_OPB_MUX_CTRL	Steuersignal des Operand B-Multiplexers der ID-Stufe im Datenpfad. Mit 0 angesteuert schaltet der Multiplexer einen Registeroperanden durch, sonst den im Kontrollpfad erzeugten Direktoperanden.
AIC_EX_ALU_CTRL(3:0)	Steuersignale der ALU, entsprechen direkt dem Opcode der arithmetisch-logischen Instruktionen.
AIC_MEM_RES_REG_EN	Enable-Signal des RES-Registers zwischen EX- und MEM-Stufe im Datenpfad.
AIC_MEM_CC_REG_EN	Enable-Signal des CC-Registers zwischen EX- und MEM-Stufe im Datenpfad.
AIC_WB_RES_REG_EN	Enable-Signal des RES-Registers zwischen MEM- und WB-Stufe im Datenpfad.
AIC_WB_CC_REG_EN	Enable-Signal des CC-Registers zwischen MEM- und WB-Stufe im Datenpfad.
AIC_WB_W_PORT_A_ADDR(3:0)	Adresse für Schreibport A des Registerspeichers, wirksam in der WB-Stufe.
AIC_WB_W_PORT_A_EN	Enable-Signal von Schreibport A des Registerspeichers.
AIC_WB_IAR_MUX_CTRL	Steuersignal des Multiplexers in der WB-Stufe, der entweder das LOAD- ('1') oder das RES-Register ('0') mit dem Dateneingang des IAR verbindet.
AIC_WB_IAR_LOAD	Lade-Steuersignal für das IAR. Verursacht wenn gesetzt einen Sprung zur Adresse, die am Dateneingang des IAR anliegt.
AIC_WB_PSR_EN	Enable-Signal des Statusregisters, für WB_PSR_EN = '0' sind alle Schreibzugriffe auf das Statusregister wirkungslos.
AIC_WB_PSR_SET_CC	Steuersignal des Statusregisters, führt wenn gesetzt zur Aktualisierung des Condition-Codes.
AIC_WB_PSR_ER	Steuersignal des Statusregisters, zeigt an, dass das SPSR des aktuellen Modus in das CPSR kopiert werden soll (was der Rückkehr aus einer Ausnahmebehandlung entspricht).
AIC_DELAY(1:0)	Zahl der Wartetakte, die die Prozessorsteuerung z.B. nach einem Sprung in einem Wartezustand verbleibt, bevor wieder eine Instruktion gefetcht oder decodiert wird.
AIC_ARM_NEXT_STATE	Zustand der Prozessorsteuerung im nächsten Takt.

Tabelle 1: Schnittstelle der Prozessor für arithmetisch-logische Instruktionen

Alle drei Befehlsgruppen können in einem Modul bearbeitet werden, da sie sehr ähnliche Auswirkungen auf den Prozessor haben. Die von Ihnen im Modul gesetzten Steuersignale werden ausschließlich dann von der Prozessorsteuerung übernommen, wenn eine arithmetisch-logische Instruktion vorliegt, Sie müssen das nicht erneut überprüfen. Das Modul verfügt dennoch über einen Eingang für das Ergebnis des Instruktionsgruppendecoders, weil für die verschiedenen Gruppen die Steuersignale minimal unterschiedlich zu setzen sind.



Die diversen Shift- und Rotationsmöglichkeiten von Operand 2 müssen von Ihnen nicht ausgewertet werden: In **ArmControlPath.vhd** ist das Modul **ArmShiftRecoder** instanziiert. Das Modul nimmt Operand 2 einer Instruktion sowie das Ergebnis des Instruktionsgruppendecoders entgegen und vereinheitlicht die diversen Shift- und Rotationsangaben. Diese Vereinfachung wurde bereits in der Dokumentation des Shifters angedeutet. Der Shift-Recoder erzeugt selbstständig alle Shifter-Steuersignale, außerdem extrahiert er bereits direkt angegebenen Schiebeweiten aus Instruktionen. Ebenfalls durch den Shift-Recoder wird das Steuersignal für den Multiplexer ID_SHIFT_MUX im Datenpfad bestimmt und davon auch das Signal ABC_INST0_SHIFT_REG_USED der Bypasssteuerung abgeleitet. Deshalb treten all diese Signale in ArmArithInstructionCtrl.vhd nicht auf.

Setzen Sie die Steuersignale aus obiger Tabelle gemäß der folgenden Anforderungen:

- Alle Steuersignale, die den Prozessorzustand verändern (also das Schreiben in Register verursachen), z.B. AIC_WB_PSR_EN und AIC_WB_W_PORT_A_EN dürfen nur gesetzt sein, wenn die Zustandsänderung explizit erwünscht ist. Andere Signale dürfen einen beliebigen (binären) Wert annehmen, falls sie ohnehin wirkungslos sind. Beispielsweise darf AIC_ID_R_PORT_C_ADDR beliebig sein, sofern Operand C nicht bei der Instruktionsausführung verarbeitet wird.
- Die Steuersignale der ALU entsprechen unmittelbar den Instruktionsbits 24:21, also dem *Opcode*-Feld.
- Der erste Operand aller arithmetisch-logischen Instruktionen wird immer benötigt, seine Registeradresse entspricht den Instruktionsbits 19:16 (Rn). Er wird aus Port A des Registerspeichers gelesen.
- Ergebnisse werden immer über den Schreibport **A** des Registerspeichers zurückgeschrieben. Die Registeradresse entspricht unmittelbar den Instruktionsbits 15:12 (Rd).
- Für Instruktionen der Gruppe `CD_ARITH_IMMEDIATE` muss der Direktoperand (Instruktionsbits 7:0) mit Nullen erweitert und in den Datenpfad eingebracht werden. Dazu ist der Multiplexer ID_OPB_MUX entsprechend zu schalten. Die Erzeugung des Direktoperanden gestaltet sich besonders einfach, wenn Sie berücksichtigen, dass er für die anderen beiden Befehlsgruppen ohnehin nicht verarbeitet wird.
- Operand B wird aus dem Registerspeicher gelesen, falls es sich um eine Instruktion aus der Gruppe `CD_ARITH_REGISTER` oder `CD_ARITH_REGISTER_REGISTER` handelt. Die entsprechende Adresse entspricht unmittelbar den Instruktionsbits 3:0 (Rm).
- Für Instruktionen der Gruppe `CD_ARITH_REGISTER_REGISTER` muss ein weiterer Registeroperand (Operand **C**) gelesen werden. Die Registeradresse entspricht den Instruktionsbits 11:8 (Rs).

- Setzen Sie für jede Instruktionsgruppe die einzelnen Bits von AIC_ID_REGS_USED entsprechend der tatsächlich aus dem Registerspeicher benötigten Operanden.
- Zeigt `Opcode` eine der Instruktionen `TST`, `TEQ`, `CMP` oder `CMN` an, so wird der veränderte Condition-Code unbedingt in das Statusregister geschrieben, der Registerspeicher bleibt aber unverändert. Setzen Sie dazu die Signale AIC_WB_PSR_EN und AIC_WB_PSR_SET_CC. Achtung: AIC_WB_PSR_ER darf nicht gleichzeitig gesetzt sein. Sorgen Sie dafür, dass die Enable-Signale der Register `MEM_CC_REG` und `WB_CC_REG` gesetzt sind. Der Wert von Rd spielt keine Rolle, das S-Bit ist für diese Opcodes immer gesetzt und muss nicht ausgewertet werden. Diese vier Opcodes verursachen unabhängig vom Wert in Rd **nie** einen Sprung.
- Für die übrigen 12 Opcodes ist folgendes Verhalten zu realisieren:
  - Das Ergebnis der ALU-Operation wird **immer** in den Registerspeicher geschrieben, dazu sind die *Write-Enable*-Signale der Register `MEM_RES_REG`, `WB_RES_REG` und des Registerspeicher-Schreibports **A** zu setzen.
  - Entspricht Rd der Adresse 15 (der PC!), so muss ein Sprung durchgeführt werden. Sie können dabei weiterhin den gewöhnlichen Schreibzugriff auf den Registerspeicher durchführen, müssen aber zusätzlich das Ladesignal des Instruktionsadressregisters setzen, wodurch das Ergebnis der Berechnung aus der Execute-Stufe in der WB-Stufe als Sprungziel in das IAR geladen wird. Das Steuersignal des Multiplexers WB_IAR_MUX muss so geschaltet sein, dass Daten aus dem RES-Pfad das IAR erreichen können.
  - Ist das S-Bit gesetzt, Rd aber nicht der PC, wird neben dem Registerspeicher auch das Statusregister (und dort nur der Condition-Code) aktualisiert. Es gilt wieder: AIC_WB_PSR_EN und AIC_WB_PSR_SET_CC müssen gesetzt werden, AIC_WB_PSR_ER darf nicht gesetzt sein.
  - Ist das S-Bit gesetzt und Rd der PC, so wird ein Sprung durchgeführt und zusätzlich ein Moduswechsel im Statusregister ausgelöst. Genauer: Es findet eine Rückkehr aus einer Ausnahmebehandlung statt, weil das SPSR des aktuellen Modus in das CPSR geschrieben wird. Sie erreichen dies durch Setzen von AIC_WB_PSR_EN und AIC_WB_PSR_ER während AIC_WB_PSR_SET_CC nicht gesetzt ist.
- Außerdem gilt: Verursacht die Instruktion keinen Sprung, ist der Nachfolgezustand der Steuerung `STATE_DECODE`. Andernfalls wechselt die Steuerung in den Zustand `STATE_WAIT_TO_FETCH`.
- Findet kein Sprung statt, erhält AIC_DELAY den Wert `00`, sonst `10` (entspricht drei Takten – 2, 1, 0 – im Wartezustand).
- Findet kein Sprung statt, soll die Instruktionsadresse im aktuellen Takt inkrementiert, AIC_IF_IAR_INC also gesetzt werden.
- **ArmControlPath** muss für diese Aufgabe nicht bearbeitet werden, lediglich das Einbinden von Simulationsmodellen kann notwendig sein (siehe unten).

Verwenden Sie die Testbench **ArmArithInstructionCtrl_tb.vhd**, um das Verhalten ihrer Implementierung zu überprüfen. Testen Sie die Implementierung abschließend im Kontext des Gesamtsystems mit der Testbench **ArmCore_tb** (nur Verhaltenssimulation).
Sie instanziiert den Prozessorkern **ArmCore.vhd**, der wiederum Datenpfad und Kontrollpfad enthält. Binden Sie dazu die zum Kontrollpfad gehörenden, vorgegebenen Module (also alle bis auf **ArmTop.vhd**) in ihr Projekt ein.
Die Testbench enthält zusätzlich ein Simulationsmodell des Arbeitsspeichers. Zu Beginn der Simulation wird ein kleiner Teil des Speichers durch die Testbench mit Instruktionen beschrieben (erkennbar am Steuersignal `INIT_RAM` in der Waveform). Anschließend führt die Testbench ein Reset durch

und übergibt die Kontrolle an den Prozessorkern. Anhand von Signalen auf den nach außen sichtbaren Leitungen (Instruktions- und Datenbus) und von Testsignalen aus **ArmGlobalProbes** wird das Verhalten des Prozessorkerns überprüft. Es handelt sich nicht annähernd um einen vollständigen Test der gesamten Funktionalität. In der Testbench werden vor allem arithmetisch-logische Instruktionen, einfache Load/Store-Instruktionen und Sprünge verwendet.

HINWEIS

Achten Sie darauf, in **ArmDataPath.vhd** und **ArmControlPath.vhd** die Simulationsmodelle aller Module einzubinden, die sie nicht fehlerfrei implementieren konnten.

Um das Simulationsmodell des ArmArithInstructionCtrl aus der ARM_SIM_LIB zu benutzen, muss eine Besonderheit beachtet werden, da in ArmArithInstructionCtrl ein selbstdefinierter Typ aus ArmTypes verwendet wird. Obwohl die Definition dieses Typen in work und ARM_SIM_LIB identisch ist, so beschwert sich Modelsim trotzdem, da es unterschiedliche Libraries sind. Um das Simulationsmodell von ArmArithInstructionCtrl zu verwenden, muss in allen(!) verwendeten VHDL-Dateien die ArmTypes aus ARM_SIM_LIB statt der aus work verwendet werden.

Ein ähnliches Problem entsteht bei der Verwendung des Registerfile-Simulationsmodelles. Da es die ArmGlobalprobes sowohl in ARM_SIM_LIB als auch in work gibt, kann die Testbench den Wert der Register nicht korrekt bestimmen. Zur Lösung muss in allen verwendeten VHDL-Dateien die ArmGlobalProbes auf ARM_SIM_LIB geändert werden.

Aufgabe 2 (1 Punkt) Compilieren, assemblieren und linken einer einfachen C-Anwendung

Als Beispielanwendung soll ein einfaches „Hallo Welt“-Programm dienen. Erzeugen Sie in ihrem ARM-Projektordner ein neues Verzeichnis, z.B. mit dem Namen **ARM_COMPILE**. Kopieren Sie die Beispielanwendung **ARM_TEST_MAIN.c** von der HWPR-Seite in diesen Ordner. Die Beispielanwendung enthält eine Endlosschleife, die permanent die Zeichen des Strings ”Hallo Welt” in das Senderegister der seriellen Schnittstelle schreibt. Vor jedem Schreibzugriff wird so lange das Statusregister der seriellen Schnittstelle gelesen, bis das Busy-Bit den Wert `0` aufweist.

Die C-Anwendung muss compiliert, assembliert und gelinkt werden. Diese Aufgaben übernehmen die verschiedenen Werkzeuge der arm-gcc-Toolchain (*arm-elf-gcc, arm-elf-as, arm-elf-ld*). Außerdem wird die gelinkte Objektdatei durch *arm-elf-objcopy* in ein anderes Binärformat (kein ELF oder COFF sondern ein vollständiges Speicherabbild der Anwendung) überführt. **Bitte beachten Sie, dass diese Tools auf den tubIT-Servern gegenwärtig nur unter Ubuntu 18 zur Verfügung stehen! Wechseln Sie für diese Aufgabe daher auf einen entsprechenden Server, z.B. mittels der Befehlszeile**

```
ssh -X ubu18.eecsit.tu-berlin.de
```

Das Ergebnis ist ein vollständiges Speicherabbild des Programms, das in den Arbeitsspeicher kopiert werden kann. Dies geschieht wieder mithilfe des System-Controllers, welchem Sie bereits in auf Aufgabenblatt 5 begegnet sind. Die diversen Bearbeitungsschritte sind bereits in einem Skript zusammengefasst. Erzeugen Sie durch Eingabe von

```
hwpti-arm-compile ARM_TEST_MAIN.c
```

in einem Terminal die Binärdatei **ARM_TEST_MAIN.c.bin** in Ihrem neuen Ordner. Kopieren Sie diese anschließend in ihren Testvektor-Ordner. Sie wird als Stimulus der folgenden Testbench benötigt.
Das Skript erzeugt durch einen Aufruf von *arm-elf-objdump* diverse Ausgaben zur erzeugten ausführbaren Datei. Interessant ist insbesondere der letzte Abschnitt (unterhalb von `Disassembly of section .text`), der Programmadressen, Maschinenbefehle (hexadezimal) und Assemblercode gegenüberstellt. Der Abschnitt ab Adresse 0x28 (<main>) entspricht der compilierten Beispielanwendung, diese wurde zu einer kleinen Assemblerdatei mit folgendem Inhalt gelinkt:

```
        .text
        .align 2                @ausrichten an Wortgrenzen
        .extern main

_start: .global _start
        b       _prog           @Sprung ueber Exceptionvektoren
        movs    pc,r14
        movs    pc,r14
        subs    pc,r14,#4
        subs    pc,r14,#8
        nop
        subs    pc,r14,#4
        subs    pc,r14,#4
_prog:  b       main            @Sprung in das gelinkte Programm
        b       _start          @Sprung nach _start
        .end
```

Dieser Code initialisiert die Exception-Vektoren an den Wortadressen 0 bis 7. In diesem Fall wird nicht in Exception-Handler gesprungen (es gibt keine) sondern unmittelbar zurück zu oder hinter die Instruktion, die eine Ausnahme verursacht hat. Dazu werden SUB- und MOV-Instruktionen verwendet. Sie haben in Aufgabe 1 eben diese Wirkung von arithmetisch-logischen Instruktionen mit gesetztem S-Bit und Rd = PC selbst implementiert.
An Adresse 0 steht der Exception-Vektor für den Reset. In diesem Fall wird nach einem Reset einfach zum Label `_prog` gesprungen, dort wiederum findet der Einsprung in das „Hallo Welt“-Programm statt.

Weil der System-Controller des ARM-Systems eine feste Programmgröße erwartet, wird die erzeugte Binärdatei schließlich mit Nullen auf 128 Byte vergrößert. Die letzte Ausgabe des Skripts gibt die Menge der angehängten Nullen an.

Aufgabe 3 (2 Punkte) Verhaltenssimulation des Gesamtsystems

Für Simulation und Synthese (Aufgabe 4) muss die Programmgröße (`ARM_PROG_MEM_SIZE` in **ArmConfiguration**) auf 128 Byte begrenzt werden. Dies verkürzt die Simulationszeit und reicht für die Beispielanwendung aus.

Kopieren Sie das neue Topmodul **ArmTop.vhd** aus den Vorgaben in ihr Projekt. Es umfasst den Prozessorkern (**ArmCore.vhd**), den Arbeitsspeicher und die übrigen Komponenten, die bereits für den praktischen Test von Aufgabenblatt 5 verwendet wurden.

Erneut müssen für die Simulation von Komponenten, die nicht fehlerfrei implementiert wurden, die entsprechenden `ARM_SIM_LIB`-Anweisungen einkommentiert werden. Passen Sie **ArmControlPath.vhd**, **ArmDataPath.vhd** und **ArmTop.vhd** entsprechend an. Denken Sie daran auch die Simulationsmodelle der vorgegebenen Komponenten **ArmProgramStatusRegister** und **ArmWordManipulation** einzubinden.

Führen Sie eine Verhaltenssimulation mit der Testbench **ArmTop_tb.vhd** durch. Die Testbench simuliert das Übertragen der Programmdatei über die serielle Schnittstelle und anschließend einige zehntausend Prozessortakte. Der System-Controller empfängt 128 Byte, schreibt diese ab Adresse 0 in den Speicher, gibt das Programm wieder aus und führt anschließend einen Reset aller anderen Komponenten durch (dabei wird der Arbeitsspeicherinhalt nicht gelöscht). Danach entkoppelt er sich vom Datenbus und übergibt dem Prozessorkern die alleinige Kontrolle.
Die Testbench protokolliert während der Laufzeit alle auf der seriellen Schnittstelle empfangenen Daten und gibt Sie nach Testende aus. Die ersten 128 Zeichen sollten dem Code des Testprogramms entsprechen, anschließend sollten Sie die Ausgaben der `Hallo Welt`-Anwendung sehen können.

Die Signale `EXT_LED(7:0)` des Topmoduls sind während der Speicherinitialisierung mit dem niederwertigen Byte der Datenbus-Adresse verbunden, anschließend mit den Bits (9:2) der Instruktionsbus-Adresse. Zur Laufzeit des Programms geben Sie also das niederwertige Byte der Wortadresse der gerade auf dem Instruktionsbus gelesenen Instruktion an. Sollten Sie Signale unterhalb des Topmoduls beobachten wollen, können sie z.B. das do-file **ArmTop_tb.udo** verwenden (und natürlich auch erweitern).

Hinweis zur Abgabe In dieser Aufgabe verwenden Sie Komponenten aus vorherigen Aufgaben. Bitte geben Sie **alle** `.vhd`-Dateien der Komponenten ab, die Sie in **ArmTop.vhd** benutzen.

Aufgabe 4 (3 Punkte) Synthese und Praxistest des Gesamtsystems

Belassen Sie `ARM_PROG_MEM_SIZE` auf 128! Kommentieren Sie alle `ARM_SIM_LIB`-Anweisungen in allen Modulen aus. Quelldateien von nicht fehlerfrei implementierten Aufgaben müssen aus dem Projekt entfernt werden und durch Netzlisten aus der `ARM_LIB` erzetzt werden (siehe Aufgabenblatt 5, Aufgabe 3). Fügen Sie dem Projekt die XDC-Datei **ArmTop.xdc** hinzu. Sie entspricht weitgehend dem Constraintfile von Aufgabenblatt 5 und muss hier nicht noch einmal von Hand erzeugt werden.

Führen Sie die Synthese von **ArmTop.vhd** durch und überprüfen Sie sorgfältig den Synthesereport auf Warnungen. Insbesondere sollten im Design keinerlei Latches oder kombinatorische Schleifen enthalten sein. Die zahlreichen *is never used*-Warnungen (auch in vorgegebenen Komponenten) können im Allgemeinen ignoriert werden.

Warnungen des Typs *internal tristates are replaced by logic* sind ausdrücklich erwünscht. Sie weisen darauf hin, dass im FPGA keine Tristate-Treiber realisiert werden können und Busse mit entsprechenden Komponenten ersetzt werden (z.B. durch Multiplexer). Notieren Sie die maximale Betriebs-

frequenz laut Synthesereport (sie sollte keinesfalls unter 10 MHz liegen). Erzeugen Sie das Bitfile und lesen Sie auch dabei alle Warnungen aufmerksam. Für die Bewertung dieser Aufgabe ist relevant, ob sich das Bitfile ohne Fehler erzeugen lässt. Der Praxistest ist dagegen wie bei Blatt 5 optional. Wollen Sie zur Rücksprache den Praxistest durchführen, senden Sie das Bitfile per E-Mail an `hardwprakt@aes.tu-berlin.de`.

HINWEIS

Sollte es im Praxistest zu Problemen kommen, die während der Verhaltenssimulation nicht aufgetreten sind, hilft evtl. eine Post Translate Simulation bei der Fehlersuche. Diese Simulation wird jedoch mehrere Stunden in Anspruch nehmen und sollte nur im Notfall durchgeführt werden. Fügen Sie also vor Beginn der Simulation alle evtl. interessanten Signale zur Waveform hinzu und machen Sie anschließend eine ausgedehnte Pause. Sie können die Simulation jederzeit unterbrechen, um den bisherigen Signalverlauf zu analysieren und sie anschließend fortsetzen.

Mündliche Rücksprache - 4 Punkte