6. Übung

Fachgebiet Architektur eingebetteter Systeme
Hardwarepraktikum (TI)

AES Embedded Systems Architecture
SoSe 17

Ausgabe: 31.05.2017

Theorie Präsentation: 07.06.2017
Praxis Abgabe: 14.06.2017

Arbeitsziel

Implementierung eines schnellen Bitaddierers, Realisierung einer kompakten Priorisierungsschaltung. Simulation mit Synthese basiertem Zeitverhalten.

Arbeitsmaterialien

- Modul `ArmRegisterBitAdder.vhd`
- Modul `ArmLdmStmNextAddress.vhd`
- Modul `ArmPriorityVectorFilter.vhd`
- Testbench `ArmRegisterBitAdder_tb.vhd`
- Testbench `ArmLdmStmNextAddress_tb.vhd`

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:

- Welche Wirkung haben die Instruktionen `LDM` und `STM`?
- Was versteht man unter einer *Self-Checking Testbench* und welche funktionalen Bestandteile weisen solche Testbenches mindestens auf?
- Beschreiben Sie die Funktionsweise und den Zweck der VHDL-*wait*-Anweisung. Gehen Sie dabei auf die *sensitivity clause* (`wait` **`on`**), die *condition clause* (`wait` **`until`**) und die *timeout clause* (`wait` **`for`**) ein.
  Nehmen Sie an, in einem Testbench-Prozess werden zwei identische *wait until*-Anweisungen mit der gleichen Bedingung hintereinander verwendet, z.B.:

  ```
  wait until test = '1';
  wait until test = '1';
  report "Bedingung erfuellt";
  ```

  Nehmen Sie weiter an, das Signal `test` sei zu Beginn der Simulation 0, sodass der Testbench-Prozess an der ersten wait-Anweisung wartet. Wird die Zeile nach der zweiten wait-Anweisung jemals ausgeführt, wenn `test` in der Simulation genau einen Pegelwechsel auf 1 durchführt und anschließend stabil bleibt? Begründen Sie Ihre Antwort.
- Welche Möglichkeiten bietet VHDL zur Ausgabe des Inhalts von Variablen oder Signalen.
- Wie können Timing Constraints eingesetzt werden um das Laufzeitverhalten einer Schaltung zu optimieren?
- Welche Vorteile bietet eine Post-Place & Route Simulation?

Empfohlene Quellen zur Bearbeitung:

- Übersicht über die ARM-Befehlssatzarchitektur [5, Kapitel 1.8.8]
- VHDL-Synthese, [6, Kapitel 6.3]

- The VHDL Cookbook, [3, Kapitel 4.2]
- Synthesis and Simulation Design Guide, [2, Kapitel 6]
- ModelSim SE User's Manual, [4, Kapitel 7]
- http://webdocs.cs.ualberta.ca/ amaral/courses/329/labs/Testbench.html

Aufgabenbeschreibung

In den folgenden Aufgaben implementieren Sie erste Komponenten für den Kontrollpfad des Prozessors. Arbeiten Sie weiterhin in Ihrem bisherigen Projekt. Beide hier entworfene Komponenten werden speziell zur Bearbeitung der Instruktionen *STM* und *LDM* benötigt

Aufgabe 1 (5 Punkte) Implementierung und Simulation eines schnellen Bitaddierers

In dieser Aufgabe soll ein, bezüglich der Geschwindigkeit optimales, Schaltnetz entwickelt werden, welches die Anzahl der Bits von einem gegebenen 16 Bit Vektor bestimmt. Die naive Lösung des Problems bestünde in der sequenziellen Addition aller Bits, faktisch also in der seriellen Verschaltung von 15 Addierern. Diese Variante weist nicht nur inakzeptable Signallaufzeiten auf, sondern bei unglücklicher Formulierung in VHDL auch einen unnötig großen Ressourcenverbrauch im FPGA.

`RBA_REGLIST(15:0)`	`IN`	Entspricht der Bitmaske der `LDM`/`STM`-Instruktionen.
`RBA_NR_OF_REGS(4:0)`	`OUT`	Zeigt im Dualcode die Zahl der in `RBA_REGLIST` gesetzten Bits (0 bis 15) an.

Tabelle 1: Schnittstelle des Bitaddierers

Realisieren Sie das Zählen der Bits stattdessen durch einen Addiererbaum im Modul **ArmRegisterBitAdder.vhd**, mit der Schnittstelle aus Tabelle 1. Dabei bestimmen kleine Addierer jeweils die Zahl der Bits in einem Ausschnitt des Bitvektors `RBA_REGLIST`. Die Ausgänge von je zwei dieser Addierer dienen als Eingänge der nächsten Addiererstufe usw. Ein einzelner abschließender Addierer erzeugt das endgültige Ergebnis.

Die Abbildung 1 verdeutlicht das Prinzip anhand eines Vektors der Breite 8. Beachten Sie, dass die Breite der Ergebnisse von Stufe zu Stufe zunimmt. Im Beispiel muss der letzte Ausgang 4 Bit breit sein, um den Wert "`1000`" annehmen zu können, wenn alle Bits des Vektors gesetzt sind.

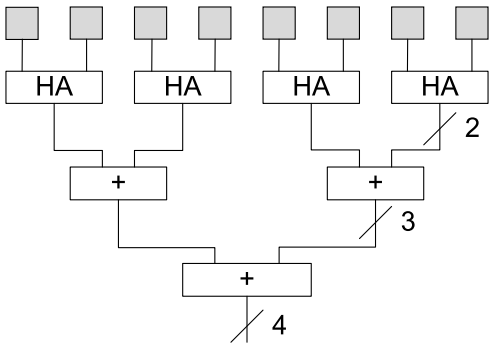

Abbildung 1: Prinzip des Addiererbaums für 8 Bit mit Halbaddierern in der ersten Stufe

Streben Sie eine Lösung an, deren längster kombinatorischer Pfad eine Laufzeit deutlich unter 10 ns (ohne Ein-/Ausgangspuffer) bzw. unter 14 ns (mit Ein-/Ausgangspuffern) aufweist. Eine grobe

Abschätzung der Signallaufzeit durch die Schaltung erhalten Sie am Ende des Synthesereports als *Maximum combinational path delay*. Ob das Modul mit Ein-/Ausgabepuffern synthetisiert wird, kann in PlanAhead in den Eigenschaften des Projektes *Project Settings* in der Rubrik Synthesis konfiguriert werden. Dazu trägt man im Feld *More Options** den Parameter `-iobuf NO` ein, wie Abbildung 2 verdeutlicht.

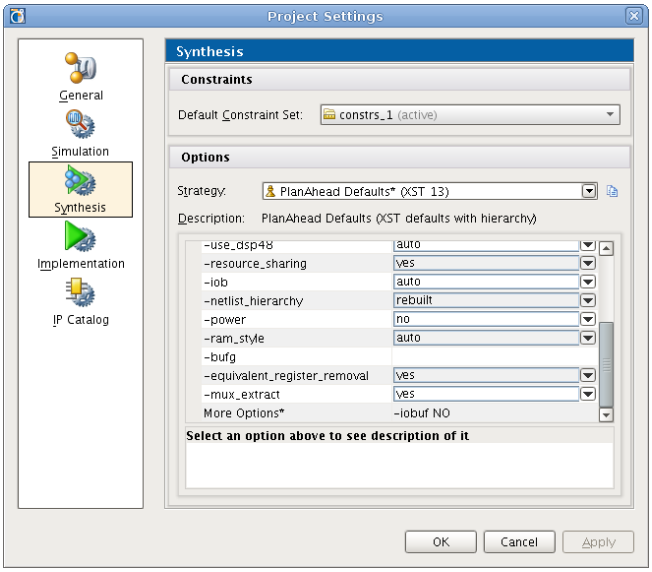

Abbildung 2: PlanAhead Synthese Einstellung für IO-Buffer

Vervollständigen Sie anschließend die Testbench **ArmRegisterBitAdder_TB** für den Funktionstest des Bitaddierers. Setzen Sie dabei mindestens folgende Anforderungen um:

- Die Testbench legt innerhalb eines Prozesses verschiedene Stimuli (Testvektoren) an das zu testende Modul an und überprüft, ob die Ausgabe des Moduls die korrekte Anzahl von Bits anzeigt.
- Legen Sie mind. 16 unterschiedliche, möglichst verschiedene Testvektoren an.
- Innerhalb der Testbench muss zu jedem Testvektor die Zahl der gesetzten Bits ermittelt werden. Sie können sie von Hand vorgeben oder hochsprachlich ermitteln (z.B. mit einer Zählschleife über dem Testvektor).
- Erzeugen Sie für jeden festgestellten Fehler eine Ausgabe auf der Konsole mittels der VHDL-Anweisungen *report* und *assert*.
- Geben Sie am Ende der Testbench an, ob der Test erfolgreich durchgeführt wurde.
- Initialisieren Sie das Stimulussignal für `RBA_REGLIST` mit 0. Legen Sie die ersten „echten“ Testwerte frühestens 100 ns nach Beginn der Simulation an.
- Warten Sie innerhalb der Testbench nach anlegen eines Testvektors mind. 14 ns und überprüfen Sie erst dann das Ergebnis. Warten Sie dann weiter 10 ns und stellen Sie sicher, dass sich das Signal *RBA_NR_OF_REGS* innerhalb dieser Zeitspanne nicht mehr geändert hat, z.B. mittels des *stable*-Attributs.

Führen Sie eine Verhaltenssimulation in Modelsim durch. Innerhalb der Verhaltenssimulation wird unmittelbar der VHDL-Code simuliert. Dabei vergeht keine simulierte Zeit zwischen Anlegen eines Stimulus und der Erzeugung von Ergebnissen im Modul.

Führen Sie anschließend eine *Post-Translate* Simulation durch. Erklären Sie dazu in PlanAhead das

Modul zum Topmodul und führen Sie dazu die Schritte Synthesize und Implement[1] aus. Leider lassen sich Simulation-Modelle nicht direkt aus PlanAhead erzeugen. Daher müssen wir diese selbständig mit Hilfe des Werkzeuges *netgen* [1, Kapitel 4] erzeugen.

Öffnen Sie dazu ein Terminal und wechseln sie in das PlanAhead Projekt-Verzeichnis. Hier sollten Sie einen Ordner `<projectname>.runs` finden. In diesem Ordner finden Sie mindestens ein Verzeichnis `impl_<nr>`. Wechseln Sie in das aktuelle Implementierungsverzeichnis. Hier sollte sich die Datei `ArmRegisterBitAdder.ngd` befinden. Mit dem folgenden Aufruf von *netgen* lässt sich aus dieser Datei ein VHDL-Modell erzeugen.

```
netgen -sim -ofmt vhdl ArmRegisterBitAdder.ngd ArmRegisterBitAdder-translate.vhd
```

Die erzeugte Datei (`ArmRegisterBitAdder-translate.vhd`) kann durch ihre Testbench instanziiert und mit *ModelSim* simuliert werden.

Auch in der *Post-Translate* Simulation existiert kein realitätsnahes Zeitverhalten. Die Simulation erfolgt jedoch nicht mehr unmittelbar auf dem von Ihnen beschriebenen Code, sondern auf Verhaltensmodellen der FPGA-Komponenten, auf die Ihr Design während der Synthese abgebildet wurde. Abweichungen zwischen intendiertem Verhalten in VHDL und Umsetzung auf Hardware durch die Synthese können hier gefunden werden.

Führen Sie schließlich eine *Post-Route* Simulation durch. In dem sie genau wie bei der *Post-Translate* Simulation vorgehen, jedoch übergeben Sie *netgen* dieses mal, die Netzliste, welche alle FPGA spezifischen Informationen enthält (`.ncd`), sowie ein *Physical Contraint File* (`.pcf`). Beachten Sie, dass der Backslash im folgenden Befehl lediglich einen Zeilenumbruch markiert und nicht mit kopiert werden darf.

```
netgen -sim -ofmt vhdl -pcf ArmRegisterBitAdder.pcf \
    ArmRegisterBitAdder_routed.ncd ArmRegisterBitAdder-timesim.vhd
```

Erzeugt wird nun nicht nur eine VHDL Netzliste sondern zusätzlich ein sdf File welche die Verzögerungen enthält. Damit dies von *ModelSim* berücksichtigt wird muss es beim Start der Simulation angegeben werden. Dies kann über den Dialog (*Simulate -> Start Simulation* ...) erreicht werden, ähnlich wie in Abbildung 3. Beachten Sie, dass bei SDF unter Region “UUT„ angegeben werden muss. Weitere Informationen finden Sie in [4, Kapitel 31].

In der ModelSim-Waveform können Sie nun nachvollziehen, dass der Ausgang des getesteten Moduls sich erst einige Zeit nach Anlegen eines neuen Stimulus ändert und dass die Änderungen der fünf Ausgangsbits zu unterschiedlichen Zeitpunkten erfolgen.

HINWEIS

Die Implementierungsdetails des Addierers bleiben Ihnen überlassen, solange ersichtlich ist, dass sie eine Baumstruktur verwenden. Die Addierer der ersten Stufe können entweder 2 Bit des Eingangsvektors verarbeiten (Halbaddierer) oder auch 3 Bit (Volladdierer). In jedem Fall sind die Ausgänge der ersten Addiererstufe 2 Bit breit. Sie können die Addierer implementieren, indem sie die Halb- bzw. Volladdierergleichungen verwenden (Summe und Übertrag) oder die VHDL-Additionsoperatoren (für höhere Addiererstufen ist dies zu empfehlen). Achten Sie darauf, dass die Addierer nicht breiter als unbedingt notwendig ausfallen.

Für die erste Stufe ist es auch gestattet, die Abbildung des Bitmusters auf den Ausgang durch eine Tabelle (*case*-Anweisung!) zu realisieren, in diesem Fall dürfen Sie bis zu 4 Bits des Bitvektors auf einmal zusammenfassen (wobei der Ausgang dann 3 Bit breit ist). Sie können, um die Übersichtlichkeit zu steigern, Module oder Funktionen zur Zählung in den Teilvektoren anlegen.

Wahrscheinlich ist ihrem Projekt noch ein UCF aus Aufgabenblatt 5 zugewiesen. Entfernen Sie das UCF aus dem Projekt (löschen Sie **nicht** die Datei) oder kommentieren Sie alle Zeilen des UCF aus (Kommentarzeichen: #).

[1] Implement ist nur mit IO-Buffern möglich.

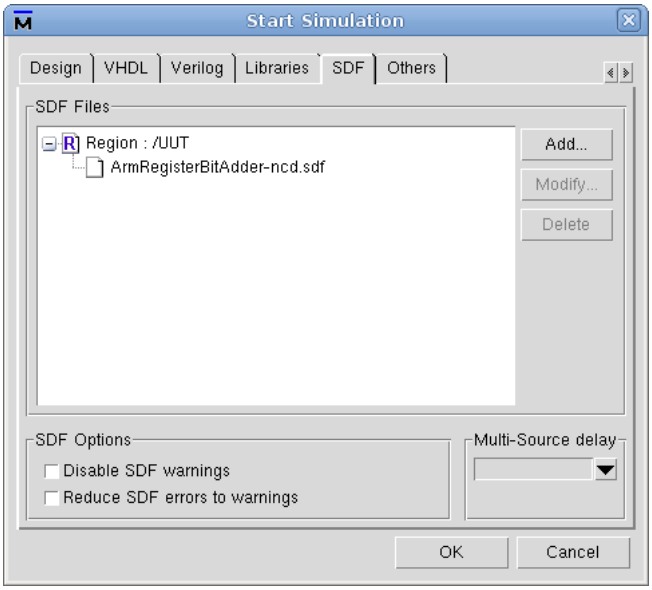

Abbildung 3: *ModelSim* Diaglog zum Start einer Simulation mit sdf-File

Während der Verhaltenssimulation war es möglich, ein verändertes Modul in *ModelSim* unmittelbar neu zu laden, da die Simulation auf dem VHDL-Code basierte. In allen anderen Simulationen muss nach einer Änderung am VHDL-Code die Synthese erneut durchgefürt werden.

Aufgabe 2 (5 Punkte) Priorisierungsschaltung und Registeradressbestimmung

In dieser Aufgabe vervollständigen Sie die Module **ArmLdmStmNextAddress.vhd** und **ArmPriorityVectorFilter.vhd**. **ArmPriorityVectorFilter** wird innerhalb von **ArmLdmStmNextAddress** instanziiert.

ArmPriorityVectorFilter.vhd ist eine Priorisierungsschaltung, ein sogenannter Prioritätsencoder, dessen Schnittstelle der Tabelle 2 entnommen werden kann.

`PVF_VECTOR_UNFILTERED(15:0)`	`IN`	Bitmaske der `LDM`/`STM`-Instruktionen
`PVF_VECTOR_FILTERED(15:0)`	`OUT`	Gefilterte Bitmaske. Maximal ein Bit ist gesetzt, alle übrigen sind '0'.

Tabelle 2: Schnittstelle des Prioritätsencoders

Der Prioritätsencoder ist eine rein kombinatorische Schaltung. Er nimmt eine 16-stellige Bitmaske entgegen, in der 0...16 Bits gesetzt sein können. War am Eingang kein Bit gesetzt, so gilt das auch für den Ausgang. War am Eingang genau ein Bit gesetzt, ist auch genau dieses Bit am Ausgang gesetzt. Sind mehrere Eingangsbits gesetzt, wird am Ausgang nur das Bit mit dem niedrigsten Index gesetzt. Der Eingangswert `X"FFFF"` wird also auf den Ausgangswert `X"0001"` abgebildet. Bits mit niedrigem Index werden gegenüber solchen mit hohem Index priorisiert.

ArmLdmStmNextAddress.vhd implementiert ein Register der Breite 16 Bit, in das im Normalfall permanent die niederwertigen 16 Bit der gerade aus dem Instruktionsspeicher gelesenen Instruktion kopiert werden. Tabelle 3 beschreibt die Schnittstelle dieser Komponente.

SYS_RST	IN	Setzt das interne 16-Bit-Register synchron auf 0.
SYS_CLK	IN	Mit steigender Taktflanke wird der Wert des internen Registers aktualisiert.
LNA_LOAD_REGLIST	IN	Ist das Steuersignal gesetzt, wird das interne Register taktsynchron mit dem Wert am Eingang LNA_REGLIST überschrieben.
LNA_HOLD_VALUE	IN	Für LOAD_REGLIST = 0 und HOLD_VALUE = 1 wird der Wert des Registers gehalten. Für LOAD_REGLIST = 0 und HOLD_VALUE = 0 wird das Bit mit der aktuell höchsten Priorität auf '0' gesetzt.
LNA_REGLIST(15:0)	IN	Ein Bitvektor, in dem eine beliebige Menge von Bits gesetzt sein kann.
LNA_ADDRESS(3:0)	OUT	Gibt im Dualcode den Index des gesetzten Bits mit der höchsten Priorität aus. (also "0000" für Bit 0 , "0001" für Bit 1 usw.) und ebenfalls "0000", wenn kein Bit gesetzt ist.
LNA_CURRENT_REGLIST_REG(15:0)	OUT	Gibt den aktuellen Zustand des internen Registers aus. An diesem Ausgang wird später der Bitaddierer aus Aufgabe 1 angeschlossen.

Tabelle 3: Schnittstelle des ArmLdmStmNextAddress

- Das Register wird mit jeder steigenden Taktflanke neu geladen, sofern das Steuersignal LNA_LOAD_REGLIST gesetzt ist.
- LNA_HOLD_VALUE ist nur für LNA_LOAD_REGLIST = 0 von Bedeutung. In diesem Fall wird der Inhalt des Registers mit der nächsten Taktflanke nicht verändert.
- Der Prioritätsencoder wird permanent durch den Inhalt des Registers gespeist und gibt das Bit mit der höchsten Priorität zurück. Diese Information wird verwendet, wenn weder ein neuer Wert geladen noch der bisherige Wert im Register gehalten werden soll. Dann ist folgendes Verhalten zu realisieren: Das vom Prioritätsencoder bestimmte Bit wird mit der nächsten steigenden Taktflanke im internen Register von **ArmLdmStmNextAddress** auf '0' gesetzt, die übrigen Bits aber nicht verändert.
- Jedem der Bits des Registers ist eine 4-Bit-Adresse zugeordnet. Sie wird nach einer Adressübersetzung in der Prozessorsteuerung zur Adressierung des Registerspeichers verwendet. Das Modul **ArmLdmStmNextAddress** gibt am Ausgang LNA_ADDRESS permanent die Registeradresse des am höchsten priorisierten, gesetzten Bits aus und "0000" wenn kein Bit gesetzt ist.
- Ist in LNA_REGLIST zum Beispiel jedes Bit gesetzt, erscheint am Ausgang LNA_ADDRESS nach dem Laden in das interne Register der Wert "0000", im anschließenden Takt der Wert "0001", dann "0010", ..., "1111" und anschließend dauerhaft "0000", bis ein neuer Wert in das Register geladen wird.

Testen Sie Ihre Implementierung mit der Testbench **ArmLdmStmNextAddress_TB**. Erklären Sie das Modul zum Topmodul. Führen Sie erst eine Verhaltenssimulation durch und anschließend eine *Post-Route* Simulation. Diese berücksichtigt sämtliche Verzögerungen auf dem FPGA inkl. der Signallaufzeiten durch Komponenten„verdrahtung". Dazu muss die Synthese zwingend mit Ein-/Ausgangspuffern durchgeführt werden.

Aufgabe 3 (2 Punkte) Ermittlung der maximalen Betriebsfrequenz des Moduls

Gehen Sie mit dem Pfeil neben *Implement* auf *Implementation Settings*. Unter *Options* können Sie nun die Implementierung konfigurieren. Um einen ausführlichen Timing Report zu erhalten müssen Sie unter *Status Timing Report (trce) -> More options* den Parameter "-a" angeben. Bestätigen Sie den Dialog mittels *OK* und *Run*. Nach Beendigung der Implementierung können Sie sich über *View Reports* den *Trace Report* anzeigen lassen.

Der erste Eintrag im geöffneten Report sollte die Bezeichnung *Timing constraints: Default period analysis for net <Name eines Clockbuffers>* tragen. Dort finden Sie einen Eintrag *Minimum Period*, dies ist die Länge des kritischen Pfads der Schaltung, dessen Kehrwert die maximale Betriebsfrequenz bestimmt. Unmittelbar darunter wird der kritische Pfad (Quelle, Ziel, durchlaufene Logik) im Detail beschrieben. Notieren Sie den kritischen Pfad der Schaltung und schließen Sie den *Timing Analyzer*. Es sei angemerkt, dass der kritische Pfad sich nur auf Logik zwischen zwei Registern bezieht. Innerhalb des Reports werden auch deutlich längere Pfade angegeben, diese beziehen sich aber auf Ausgänge des Designs unter der Annahme, dass **ArmLdmStmNextAddress** der einzige Inhalt des FPGA ist.

Mündliche Rücksprache - 4 Punkte

Literatur

[1] *Command Line Tools User Guide*. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/devref.pdf`, Oktober 2011.

[2] *Synthesis and Simulation Design Guide*. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/sim.pdf`.

[3] ASHENDEN, PETER J.: *The VHDL Cookbook*. First Edition Auflage, Juli 1990. URL `http://tams-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf`.

[4] MENTOR GRAPHICS CORPORATION: *ModelSim SE User's Manual*, 6.6d Auflage.

[5] RECHNERTECHNOLOGIE, TU BERLIN FG: *Hardwarepratikum Technische Informatik Übersicht über die ARM-Befehlssatzarchitektur*, Oktober 2010.

[6] REICHARDT, JÜRGEN und BERND SCHWARZ: *VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme*. Oldenbourg, 4., überarbeitete Auflage. Auflage, Oktober 2007. ISBN 3486581929.