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

Ausgabe: 25.05.2020

Theorie Präsentation: 08./09.06.2020
Praxis Abgabe: 21.06.2020

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`

Abgabedateien

- Datei `ArmRegisterBitAdder.vhd`
- Datei `ArmRegisterBitAdder_tb.vhd`
- Datei `ArmLdmStmNextAddress.vhd`
- Datei `ArmPriorityVectorFilter.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 (auch Post-Implementation) Simulation?

Empfohlene Quellen zur Bearbeitung:

- Übersicht über die ARM-Befehlssatzarchitektur [3, Kapitel 1.8.8]
- VHDL-Synthese, [4, Kapitel 6.3]
- The VHDL Cookbook, [1, Kapitel 4.2]
- Synthesis and Simulation Design Guide, [5, Kapitel 6]
- ModelSim SE User's Manual, [2, Kapitel 6]
- 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.

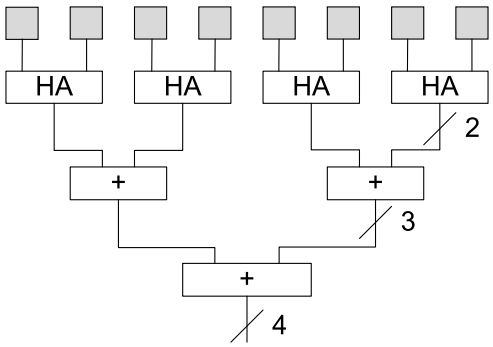


Abbildung 1: Prinzip des Addiererbaums für 8 Bit mit Halbaddierern in der ersten Stufe

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. Volladdiererergleichungen 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 XDC aus Aufgabenblatt 5 zugewiesen. Entfernen Sie das XDC aus dem Projekt (löschen Sie **nicht** die Datei) oder kommentieren Sie alle Zeilen des XDC aus (Kommentarzeichen: #).

Während der Verhaltenssimulation war es möglich, ein verändertes Modul in *QuestaSim* 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.

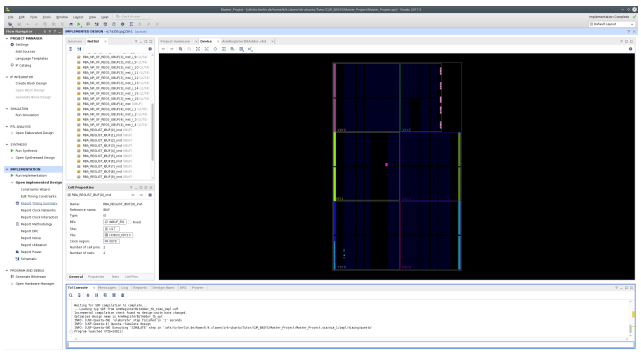
Streben Sie eine Lösung an, deren längster kombinatorischer Pfad eine Laufzeit **unter 11 ns** aufweist. Wie der längste kombinatorische Pfad bestimmt werden kann, wird im Folgenden gezeigt.

Als erstes muss `ArmRegisterBitAdder` als *Top-Modul* ausgewählt und *implementiert* (nicht nur synthetisiert !) werden. Nachdem die Implementation abgeschlossen ist, öffnet man das *Implemented Design*; entweder man klickt unter dem Reiter *Implementation* drauf oder man wählt es im Fenster direkt aus, welches sich automatisch nach der Implementation öffnet. Nun stehen im Reiter unter *Open Implemented Design* weitere Optionen zur Auswahl. Drücken Sie auf *Report Timing Summary*, wie es in Abb. 2a zu sehen ist.

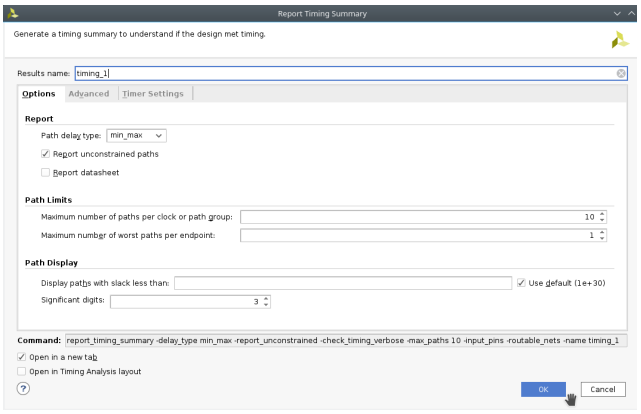
Ändern Sie nichts an den Standardeinstellungen und klicken Sie auf *OK* (siehe Abb. 2b). Daraufhin öffnet sich ein neuer Tab neben der *Tcl Console*. Scrollen Sie nun nach ganz unten und wählen *Setup*, wie folgt aus:

- ▽ Unconstrained Paths
  - ▽ NONE to NONE
    - ▷ Setup (5)

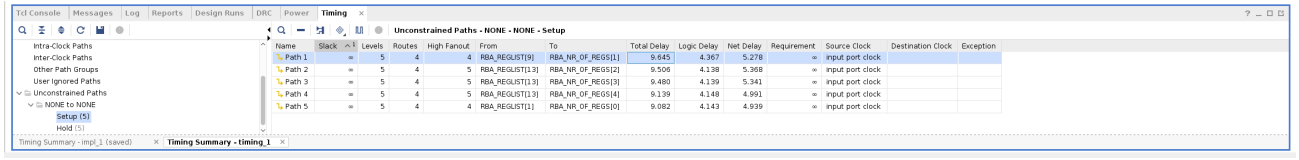
Das Fenster auf der rechten Seite zeigt die längsten Pfade im Top-Modul an. Die erste Zeile zeigt dabei den längsten Pfad an. Achten Sie darauf, dass der *Total Delay* **unter 11 ns** liegt, so wie in Abb. 2c.



(a) Öffnen von *Report Timing Summary*



(b) *Timing Summary* Einstellungen



(c) *Total Delay* bestimmen

Abbildung 2: Bestimmung des längsten Pfades

HINWEIS

Wie bereits einige vermuten können, ist dies nicht der *empfohlene Weg*, wie man kritische Pfade unter Vivado analysieren sollte. Korrekt wäre es, wenn man sich zu dem jeweiligen Top-Modul eine XDC-Datei anlegt und direkt die *Timing Constraints* angibt, welche man einhalten möchte. Diese könnte direkt in der Synthese/Implementation automatisch ausgewertet werden. Da hier jedoch der längste *kombinatorische* Pfad untersucht wird, also der längste Weg ohne Flip-Flops als Quelle und Ziel, müsste man noch eine virtuelle Clock hinzufügen und gewisse Delays abschätzen. Dies würde aber den Rahmen des Aufgabenblattes sprengen und durch diese *simple* Variante gelangt man auch auf eine akzeptable geschätzte Zeit.

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. 11 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 QuestaSim 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-Implementation* Simulation durch. Hierfür **muss** Vivado in dem Ordner gestartet werden, indem Sie normalerweise `vsim&` ausführen. Den richtigen Ordner erkennen Sie daran, dass sich in diesem eine Datei namens `modelsim.ini` befindet. Diese Datei wird im Folgenden zwingend gebraucht! Falls Sie sich nicht mehr sicher sind, in welchem Ordner man Vivado gestartet hat, starten Sie Vivado im richtigen Ordner neu, oder bewegen Sie sich mit `cd` dort hin.

Bei der *Post-Implementation* Simulation wird aus dem VHDL Code eine NGD Netzliste erzeugt. Weiterhin wird ein SDF-File erzeugt, indem festgehalten wird wie lange es dauert bis bei einem neuen Eingangssignal das Ausgangssignal aktualisiert wird. Im Gegensatz zu einer *Post-Synthesis* Simulation (die im Weiteren nicht betrachtet wird), die eine NGC Netzliste erzeugt, besitzt eine NGD Netzliste mehr Informationen über weitere Delays, wie z.B. Switching Delays.

Bevor jedoch die Simulation gestartet werden kann, muss QuestaSim als Simulator in Vivado ausgewählt werden, da eine *Post-Implementation* Simulation nur aus Vivado heraus gestartet werden kann. Gehen Sie wie folgt vor: Klicken Sie auf *Settings* im *Flow Navigator* unter *Project Manager* und wählen Sie im Settings-Fenster unter dem *Simulation*s-Reiter als *Target simulator: Questa Advanced Simulator* aus, siehe Abb. 3a. Übernehmen Sie die Einstellungen.

Dies hat zur Folge, dass statt dem Vivado Simulator nun QuestaSim gestartet wird, wenn man im *Flow Navigator* auf *Run Simulation* klickt. Damit dies funktioniert und alle Bibliotheken gefunden werden, die in QuestaSim genutzt werden, liest Vivado die `modelsim.ini` aus.

Damit auch die ArmRegisterBitAdder-Testbench gestartet wird, muss man diese unter Vivado als *Simulation Source* hinzufügen. Dabei gehen Sie wie bei dem Hinzufügen von *Design Sources* vor, wählen jedoch im ersten Schritt stattdessen *Simulation Sources* aus, siehe Abb. 3b.

Im *Hierarchy*-Fenster, indem Sie normalerweise das Top-Modul zur Syntehse auswählen, müssen Sie nun unter *Simulation Sources* `ArmRegisterBitAdder_tb` als Top-Modul auswählen. Das Top-Modul beschreibt in diesem Kontext, welche Testbench beim Klicken auf *Start Simulation* geöffnet wird.

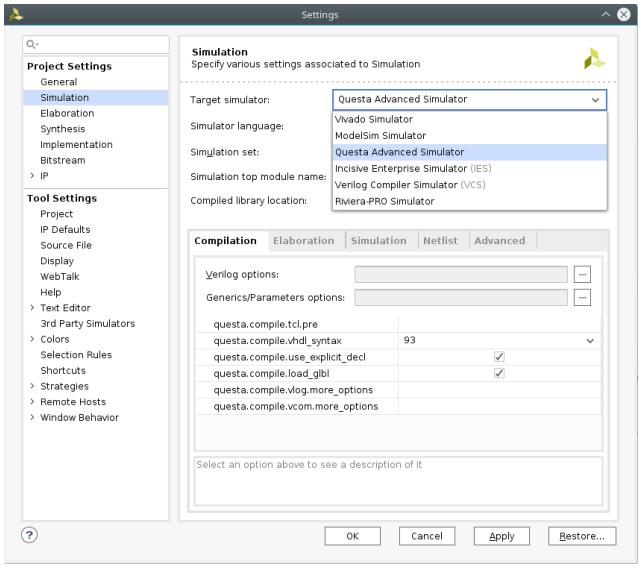
Wie der Name *Post-Implementation* Simulation schon andeutet, muss erst das Top-Modul implementiert werden, danach steht die *Post-Implementation* Simulation unter Vivado zur Verfügung. Starten Sie nun die *Post-Implementation Timing Simulation*, wie in Abb. 3c zu sehen. Falls Sie VHDL als *Simulator Language* eingestellt haben, wird sich noch ein Fenster öffnen, welches einfach bestätigt werden kann.

Daraufhin öffnet sich automatisch QuestaSim und es wird eine Simulation mit einer Laufzeit von 1000 ns gestartet. Falls Ihre Testbench länger dauert, geben Sie wie gewohnt im *Transcript*-Fenster `run -all` ein. Vivado kümmert sich dabei darum, dass das SDF-File auch in QuestaSIM geladen wird. [1]

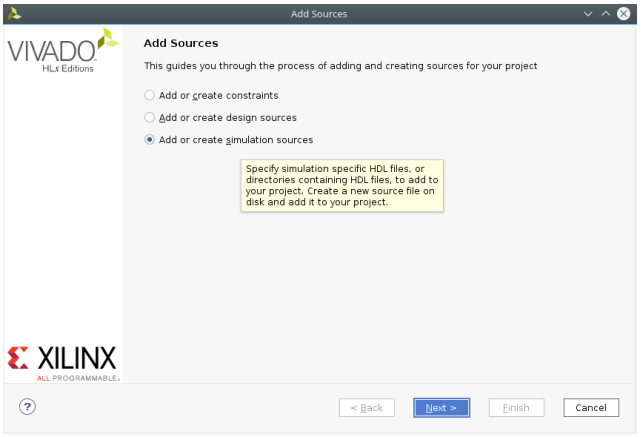
[1] Wir empfehlen es nicht QuestaSim immer aus Vivado heraus zu starten, da es besonders mit den `work` und

In der QuestaSim-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.

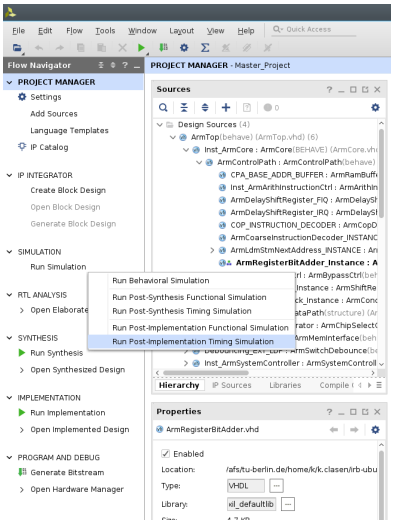
Machen Sie sich klar, warum die Testbench trotz der Verzögerung funktioniert!



(a) Einstellen der *Simulation Settings*



(b) Hinzufügen der Testbench



(c) Starten der *Post-Implementation* Simulation

Abbildung 3: *Post-Implementation* Einstellungen

`ARM_SIM_LIB` Bibliotheken zu Problemen kommen wird, wenn man nicht besondere Einstellungen unter Vivado beachtet. Das Starten aus Vivado heraus sollte nur für diese Aufgabe genutzt 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**.

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

Gehen Sie ähnlich wie in Aufgabe 1 vor und bestimmen Sie den längsten Pfad. Berechen Sie mit Hilfe des *Delays* die maximale Betriebsfrequenz des Moduls. Dies muss nicht gesondert abgegeben werden, sondern halten Sie die maximale Betriebsfrequenz für die Rücksprache bereit.

Mündliche Rücksprache - 4 Punkte

Literatur

[1] Peter J. Ashenden. *The VHDL Cookbook*. First Edition. Juli 1990. URL: `https://www.ics.uci.edu/~alexv/154/VHDL-Cookbook.pdf`.

[2] *ModelSim SE User's Manual*. 10.2c. Mentor Graphics Corporation.

[3] TU Berlin FG Rechnertechnologie. *Hardwarepratikum Technische Informatik Übersicht über die ARM-Befehlssatzarchitektur*. Okt. 2010.

[4] Jürgen Reichardt und Bernd Schwarz. *VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme*. 4., überarbeitete Auflage. Oldenbourg, Okt. 2007. ISBN: 3486581929.

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