SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Lucas Biehl

Systemarchitektur SS 2020 – Projekt

Hardware-Design mit Verilog

Abgabemodalitäten

Das Projekt beginnt am 17. Juni 2020 mit Herausgabe dieser Beschreibung. Es besteht aus zwei Teilen: einem Aufwärmteil und einem Hauptteil. Der Aufwärmteil dient dazu, dass Sie sich in Verilog als Sprache und in die verwendeten Programme einarbeiten können. Wir empfehlen, das Projekt *so bald wie möglich* zu beginnen.

Sie müssen das Projekt in *Gruppen von zwei bis drei Personen* bearbeiten. Wenn Sie eine Gruppe gebildet haben, laden Sie bitte in unserem CMS bis zum

23. Juni, 23:59 Uhr

eine Textdatei mit bis zu drei Matrikelnummern (komma-separiert) hoch. Eine 3er-Gruppe lädt folglich eine Textdatei nach genau dem folgendem Schema hoch:

```
Matrikelnummer1,Matrikelnummer2,Matrikelnummer3
```

Eine Person pro Gruppe muss die Ergebnisse *beider* Teile zusammen bis zum

Sonntag, dem 05. Juli 2020, 23:59 Uhr

in unserem CMS-System hochladen. Beide Teile fließen in die Bewertung des Projekts ein. Insgesamt sind 32 Punkte (plus 6 Zusatzpunkte) zu erreichen. Zu spät abgegebene Projekte werden mit **0 Punkten** bewertet. Verwenden Sie zur Abgabe eine ZIP-Datei oder ein gzip-komprimiertes tar-Archiv (d.h. `*.tar.gz`). Das Archiv soll unmittelbar alle Verilog-Dateien enthalten. Verwenden Sie die von uns zur Verfügung gestellten Gerüst-Dateien, welche die nötigen Verilog Modul-Deklarationen bereits beinhalten. Verändern Sie diese Modul-Deklarationen *nicht*, es sei denn es ist ausdrücklich erlaubt. Ferner verändern Sie nicht die bestehenden Dateinamen. Sind nicht alle Voraussetzungen erfüllt, kann Ihr Projekt nicht gewertet werden.

Verwenden Sie die beiden Sanitizer-Testbenches aus dem CMS um Ihr Design auf unrechtmäßige Änderungen zu prüfen. Führen Sie dazu folgendes Kommando aus:

```
iverilog -s SanitizerAufwaermteil *.v
```

(analog für den Hauptteil). Kompiliert alles ohne Probleme und Warnungen, haben wir keine Verletzung feststellen können. Stellt unser Test eine Verletzung fest, passen Sie Ihr Design für die Abgabe entsprechend an.

Fügen Sie dem Archiv außerdem eine Datei `beitraege.txt` hinzu, die kurz beschreibt, welchen Beitrag die einzelnen Teammitglieder zur Implementierung geleistet haben. Wir behalten uns vor, das Projekt für einzelne Mitglieder mit 0 Punkten zu bewerten, wenn diese keinen signifikanten Beitrag geleistet haben.

*Hinweise: Eine Zusammenarbeit mit Personen, die nicht zur eigenen Gruppe gehören, ist **nicht** erlaubt. Wir werden alle Abgaben auf Plagiate prüfen. Als Plagiate gelten auch Abgaben, die durch Modifizieren eines anderen Projekts entstanden sind, wie z.B. durch Ändern von Variablennamen. Plagiierte Abgaben werden wir mit **0 Punkten** bewerten und als Täuschungsversuch an den Prüfungsausschuss melden. Dies kann ggf. zur Exmatrikulation führen!*

*Falls Sie in der Vergangenheit schon einmal an der Systemarchitektur-Vorlesung teilgenommen haben und Ihre damalige Abgabe als Grundlage für das aktuelle Projekt verwenden möchten: Dies ist nur zulässig für Teile, die Sie selbst implementiert haben; Code, der von anderen Mitgliedern des damaligen Teams geschrieben wurde, darf **nicht** wiederverwendet werden. Fügen Sie in diesem Fall zu der Abgabe eine Datei `vorjahr.txt` hinzu, die genau beschreibt, welche Teile übernommen worden sind. Beachten Sie jedoch, dass das aktuelle Projekt nicht identisch zu den Projekten aus den Vorjahren ist. Abgaben, die zur Aufgabenstellung eines alten Projekts gehören, werden wir mit **0 Punkten** bewerten.*

Werkzeuge und Dokumentation

Zur Synthese und Simulation von Verilog-Code verwenden wir *Icarus Verilog*[1] und zum Betrachten von generierten Waveforms *gtkwave*[2]. Beide Programme funktionieren prinzipiell unter Linux, Mac OS X, und Windows. Detaillierte Installationsanweisungen finden Sie im CMS unter "Zusatzmaterial".

Zur Synthese eines Top-Level Moduls M, dessen Definition inklusive aller Sub-Module sich auf Dateien `file1.v` bis `filen.v` verteilt, rufen Sie in der Kommandozeile

`iverilog -s` M `-o sim file1.v ... filen.v`

auf. Um die Simulation zu starten, führen Sie das generierte Binary `sim` aus. Je nach Testbench sehen Sie Ihre Ergebnisse auf der Kommandozeile oder finden eine generierte Waveform-Datei, welche Sie mit `gtkwave` anschauen können.

Eine gute und sehr ausführliche Einführung in Verilog mit vielen Beispielen bietet die Seite Asic-World[3]. Informationen zur Benutzung von Icarus Verilog finden Sie unter `http://iverilog.wikia.com/wiki/Getting_Started` und für GTKWave unter `http://gtkwave.sourceforge.net/gtkwave.pdf`.

Während des Hauptteils benötigen Sie detaillierte Informationen zum MIPS-Befehlssatz, insbesondere bezüglich der Kodierung von Befehlen. Diese Informationen finden Sie auf der offiziellen Seite von MIPS[4].

Um MIPS-Programme auf Ihrem Prozessor ausführen zu können (z.B. für Testbenches) benötigen Sie das jeweilige Maschinenprogramm. Sie können den MARS-Simulator[5] verwenden um MIPS-Assembler-Programme zu schreiben und in Maschinendarstellung zu übersetzen. Verwenden Sie dazu `File->Dump Memory` und wählen Sie `Hexadecimal text` aus. Um kompatibel zu Verilogs `readmemh` zu sein, verwenden Sie bitte unsere angepasste Version[6].

Fragen und Probleme

Sollten Sie während der Umsetzung des Projekts auf Unklarheiten oder Probleme stoßen, die Sie in der Gruppe nicht lösen können, stehen wir ihnen gerne im Forum oder zu den Office Hours (Di 12.15-13.45, Mi 14.15-15.45) zu Verfügung. Generell erwarten wir jedoch, dass Sie sich bereits in Eigenarbeit (eigene Tests, Debugging Ausgaben, Online-Recherche, etc.) mit der Lösung des Problems beschäftigt haben. Unspezifische Fragen wie "Kompiliert der Code?", "Ist das richtig?" oder "Was muss in der Aufgabe XYZ gemacht werden?", werden nicht beantwortet.

Aufwärmteil

Beginnen Sie *so bald wie möglich* mit diesem Teil des Projekts, damit Sie genügend Zeit für den Hauptteil übrig haben. Die Gerüstdateien für das Projekt finden Sie in unserem CMS unter Materialien[6].

Aufgabe 1.1: Mustererkennung — 2 Punkte

Sie haben bereits auf dem Übungsblatt Mealy-Automaten zur Mustererkennung kennengelernt. In einer Sequenz von Zeichen aus dem Eingabealphabet $\{0, 1\}$ möchten wir die Muster 111 und 001 erkennen. Die Ausgabe `o[1:0]` stammt aus dem Ausgabealphabet $\{0, 1\} \times \{0, 1\}$. Das erste Bit `o[1]`/zweite Bit `o[0]` der Ausgabe soll genau dann 1 sein, wenn die beiden vorherigen Eingaben zusammen mit der aktuellen Eingabe das Muster 111/Muster 001 bilden. Hat die Maschine zum Beispiel das erste Muster 001 erkannt, so soll die Ausgabe also `o[0]` $= 0$ und `o[1]` $= 1$ sein.

Implementieren Sie eine solche Mealy-Maschine als Verilog-Modul `MealyPattern`. Schreiben Sie einen Testbench `MealyPatternTestbench`, der die Korrektheit Ihrer Konstruktion für die Sequenz 1110011001 validiert.

[1] `http://iverilog.icarus.com`
[2] `http://gtkwave.sourceforge.net`
[3] `http://www.asic-world.com/verilog/veritut.html`
[4] `https://www.mips.com/?do-download=the-mips32-instruction-set-v6-06`
[5] `http://courses.missouristate.edu/kenvollmar/mars/`
[6] `https://cms.sic.saarland/sysarch20/materials/`

Aufgabe 1.2: Divisionsschaltwerk — 5 Punkte

In der Vorlesung haben wir Schaltkreise zur Addition, Subtraktion und Multiplikation gesehen, aber nicht für die Division. Die ganzzahlige Division von zwei vorzeichenlosen Binärzahlen lässt sich, anhand der schriftlichen Division aus der Schule, als *Schaltwerk* realisieren. Die Division $\frac{\langle A \rangle}{\langle B \rangle}$ nach der Schulmethode lässt sich algorithmisch wie folgt ausdrücken.

```
R = 0
for i = N-1 to 0
  R' = 2 * R + A[i]
  if (R' < B) then Q[i] = 0, R = R'
              else Q[i] = 1, R = R'-B
```

Aufgabe Vollziehen Sie die Vorgehensweise anhand von $\frac{7}{3}$ nach.

Nun überlegen Sie sich das zugehörige Schaltwerk. Das Divisionsschaltwerk hat zwei 32-Bit Eingaben A und B, einen 1-Bit Eingang $start$, einen Eingang $clock$ und zwei 32-Bit Ausgaben Q und R, wobei Q der Quotient und R der Rest ist. 32 Takte nachdem an einer steigenden Taktflanke $start = 1$ war, soll $\langle Q \rangle$ der Quotient und $\langle R \rangle$ der Rest der Division $\frac{\langle A \rangle}{\langle B \rangle}$ sein. Wird während der Berechnung einer Division nochmals $start = 1$, so bricht das Werk die aktuelle Berechnung ab und beginnt mit den neuen aktuellen Operanden von vorne. Im nächsten Abschnitt geben wir Ihnen einige zusätzliche Hinweise.

Implementieren Sie Ihr erstelltes Schaltwerk als Verilog-Modul `Division` und überprüfen Sie Ihr Design mithilfe von Testbenches.

Hinweise Der Schaltkreis des Werkes soll pro Zyklus zwischen zwei aufeinander folgenden steigenden Taktflanken jeweils *eine* Iteration der Schleife ausführen, d.h. im Wesentlichen eine Subtraktion und einen Negativtest. Multiplikation in Hardware ist teuer. Versuchen Sie daher die Multiplikation durch günstigere Verschiebungen auszudrücken.

Verwenden Sie als Zustand des Werks drei 32-Bit breite Register: Das erste Register speichert den aktuellen Wert des Rests R. Das zweite Register speichert den aktuellen Wert des Divisors B. Das dritte Register speichert die noch benötigten Bits des Dividenden A und die bereits berechneten Bits des Quotienten Q. Das dritte Register hat also stets vor Iteration i den Zustand

$$\{A[i:0], Q[N-1:i+1]\}.$$

Ist an einer steigenden Taktflanke von $clock$ das Startsignal gesetzt ($start = 1$), so übernehmen wir die Eingaben A und B in die jeweiligen Register und beginnen mit der Berechnung. Eine Berechnung dauert exakt 32 Takte: danach liegt das korrekte Ergebnis solange an den Ausgängen an, bis eine erneute Division startet. Um zu erreichen, dass *nicht in jedem* Takt eine Iteration ausgeführt wird, sondern nur in den ersten 32 Takten nach dem $start$, kann es nützlich sein einen Zähler zu verwenden.

Tabelle 1: Steuerbits und Funktionsweise der arithmetisch-logischen Einheit. Das Verhalten für nicht aufgeführte Belegungen ist undefiniert.

$alucontrol[2:0]$			`result[31:0]`
0	0	0	$0^{31}(\langle a\rangle < \langle b\rangle \ ?\ 1 : 0)$
0	0	1	$\langle a\rangle - \langle b\rangle$
1	0	1	$\langle a\rangle + \langle b\rangle$
1	1	0	$a \mid b$
1	1	1	$a \,\&\, b$

Hauptteil

Die Gerüstdateien für den Hauptteil des Projekts finden Sie in unserem CMS unter Materialien[6]. Zur Simulation verwenden Sie bitte jeweils das Modul `ProcessorTestbench`. Wir stellen Ihnen einige Testprogramme zur Verfügung, die Sie jeweils im Testbench einkommentieren müssen. Achten Sie darauf alle diese Tests erfolgreich zu bestehen.

Sie sollten zusätzlich eigene Testbenches bauen, um die korrekte Funktionsweise Ihrer Schaltungen nachzuvollziehen. Überlegen Sie sich geeignete Testbenches und das erwartete Resultat *bevor* Sie mit der Implementierung anfangen. Ihre Testbenches für diesen Teil des Projekts gehen nicht in die Bewertung mit ein.

Aufgabe: Einzeltakt-MIPS Implementierung — 0 Punkte

Machen Sie sich mit der (fast vollständigen) Verilog-Implementierung der Einzeltaktmaschine aus der Vorlesung vertraut. Die Maschine unterstützt bisher die Instruktionen `addu`, `subu`, `and`, `or`, `sltu`, `lw`, `sw`, `addiu`, `beq` und `j`. Auch wenn diese Aufgabe keine Punkte gibt, nehmen Sie sich Zeit dafür: Haben Sie den Aufbau des Daten- und Kontrollpfades verstanden, fällt die Bearbeitung der folgenden Aufgaben wesentlich leichter.

Aufgabe 1.3: Arithmetic Logic Unit — 5 Punkte

Implementieren Sie das Modul `ArithmeticLogicModul` im Datenpfad und vervollständigen Sie die zugehörigen Steuerbits in der Dekodiereinheit gemäß Tabelle 1. Der 1-Bit Ausgang `zero` ist stets genau dann 1, wenn das Ergebnis der ALU `result[31:0]` null (0^{32}) ist.

Aufgabe 1.4: Konstanten laden — 4 Punkte

Um 32-Bit Konstanten zu laden, sind die beiden Instruktionen `lui` und `ori` sehr hilfreich. Schlagen Sie deren Kodierung und Funktionsweise in der Dokumentation des MIPS-Befehlssatzes nach. Erweitern Sie den Datenpfad und den Dekodierer entsprechend, um diese Befehle zu implementieren. Versuchen Sie die bisherige Schnittstelle zwischen Datenpfad und Dekodierer möglichst minimal zu ändern.

Tipp: Es kann sinnvoll sein, die folgenden Aufgaben bereits im Blick zu haben, wenn Sie die Schnittstelle zwischen Dekodierer und Datenpfad anpassen.

Aufgabe 1.5: Verzweigungen — 3 Punkte

Implementieren Sie die Verzweigungsinstruktion `bltz`. Versuchen Sie mit der bisherigen Schnittstelle zwischen Datenpfad und Dekodierer auszukommen.

Aufgabe 1.6: Multiplikation — 7 Punkte

Implementieren Sie die vorzeichenlose Multiplikation des MIPS-Befehlssatzes, genauer gesagt den Befehl `multu`. Überlegen Sie sich in welchem Modul des Datenpfads die Zielregister `HI` und `LO` platziert werden sollten. Machen Sie sich klar, was nun der logische Zustand einer MIPS-Maschine mit Multiplikationsfunktion ist.

Um das Ergebnis verarbeiten zu können, werden ferner die Befehle `mflo` und `mfhi` benötigt. Implementieren Sie diese beiden Befehle. Versuchen Sie die bisherige Schnittstelle zwischen Datenpfad und Dekodierer möglichst minimal zu ändern.

Aufgabe 1.7: Funktionsaufrufe — 6 Punkte

Um Funktionsaufrufe effizient zu unterstützen, benötigt man ein sogenanntes *Linkregister* um die Rücksprungadresse zu speichern. Diese wird benötigt, um am Ende eines Funktionsaufruf zum Aufrufer zurückzukehren. Die Konvention bei MIPS-Maschinen ist es, Register 31 zu verwenden. Im Assembler wird es daher auch mit ra (*return address*) bezeichnet.

Implementieren Sie die Befehle `jal` und `jr` für Funktionsaufrufe und Rücksprünge. Versuchen Sie die bisherige Schnittstelle zwischen Datenpfad und Dekodierer möglichst minimal zu ändern.

Hinweis: MIPS verwendet sogenannte *branch delay slots*. Dies bedeutet, dass bei einer Verzweigungs- oder Sprunginstruktion auch der nachfolgende Befehl ausgeführt wird bevor der Sprung tatsächlich stattfindet. Daher ist in der MIPS-Dokumentation als Wert des Linkregisters `PC + 8` aufgeführt. Wir betrachten für dieses Projekt allerdings keine delay slots und somit soll der `jal`-Befehl `PC + 4` in das Linkregister schreiben.

Aufgabe 1.8: Bonus: Division 1 — 2 Bonuspunkte

Implementieren Sie den `divu` MIPS-Befehl und verwenden Sie Ihr Divisionsschaltwerk aus dem Aufwärmteil. Die Division benötigt daher 32 Takte. Während dieser Zeit gelten die Werte des `LO` und `HI` Registers als nicht vorhersagbar.

Verwenden Sie den `divu`-Befehl von Seite 171 aus der verlinkten MIPS-Dokumentation. Diese beschreibt eine ältere Variante der Division, die das Ergebnis in `HI` und `LO` speichert. Neuere MIPS-Varianten schreiben den Quotienten direkt in ein General-Purpose Register.

Hinweis: Verwenden Sie die Register `LO` und `HI` *clever*, d.h. überlegen Sie welche Speicherfunktion im Schaltwerk sie übernehmen können.

Es existiert eine Abhängigkeit zwischen dem *Lesen* eines `mflo/hi`-Befehls und dem *Schreiben* eines vorangegangenen `divu`-Befehls. Der Divisionsbefehl benötigt mehrere Takte zur Berechnung des korrekten Resultats. In der Zwischenzeit arbeitet die Einzeltakt-Maschine bereits die folgenden Instruktionen ab. Dadurch wird die obige Lese-Schreib-Abhängigkeit problematisch: Es hängt nun von der Anzahl der Instruktionen zwischen `mflo/hi` und `divu` ab, ob `mflo/hi` das korrekte Resultat oder einen unvorhersagbaren Wert liest. Eine solche Situation nennt man daher auch *hazard*. Um obigem *hazard* zu entgehen, sollte man nach einer Division 32 Takte lang kein `mflo` oder `mfhi` ausführen. Für die Einhaltung dieser Konvention, auch *Software Condition* genannt, muss der Programmierer beziehungsweise der Compiler sorgen.

Aufgabe 1.9: Bonus: Division 2 — 4 Bonuspunkte

Wir möchten obige Software Condition loswerden, um dem Programmierer das Leben zu erleichtern. Statt den oben beschriebenen *hazard* in Software aufzulösen, kann man solche Situationen auch in Hardware lösen. Dazu verwendet man einen sogenannten *interlock* (Verriegelung): Wenn man erkennt, dass die aktuelle Instruktion das `HI` oder `LO` Register zugreift während eine Division ausgeführt wird, so "hält" man die Ausführung der Instruktion "an".

Überlegen Sie sich, wie ein solches "Anhalten" implementiert werden kann. Erweitern Sie Ihr Divisionsschaltwerk um einen Ausgang `busy`, der 1 ist während eine Division ausgeführt wird. Implementieren Sie einen Interlock-Mechanismus für obige Situation.