SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Lucas Biehl

Systemarchitektur SS 2021 – Projekt 1

Hardware-Design mit Verilog

Abgabemodalitäten

Das Projekt beginnt am 9. Juni 2021 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 dürfen das Projekt in *Gruppen von zwei bis drei Personen* bearbeiten. Wenn Sie eine Gruppe gebildet haben, erstellen Sie bitte auf Ihrer persönlichen Statusseite im CMS bis zum

Montag, dem 14. Juni 2021, 23:59 Uhr

ein entsprechendes Team.

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

Montag, dem 28. Juni 2021, 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 (d.h. verwenden Sie keine Unterordner). 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[12].

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 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 X.Y 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 101 und 010 erkennen. Die Ausgabe `o[1:0]` stammt aus dem Ausgabealphabet $\{0, 1\} \times \{0, 1\}$. Das linke Bit `o[1]`/rechte Bit `o[0]` der Ausgabe soll genau dann 1 sein, wenn die beiden vorherigen Eingaben zusammen mit der aktuellen Eingabe das Muster 101/Muster 010 bilden. Hat die Maschine zum Beispiel das Muster 101 erkannt, so soll die Ausgabe 10 sein. Zu Beginn (d.h. bevor ausreichend viele Eingaben gelesen worden sind), soll die Ausgabe 00 sein.

Implementieren Sie eine solche Mealy-Maschine als Verilog-Modul `MealyPattern`.

Schreiben Sie einen Testbench `MealyPatternTestbench`, der die Korrektheit Ihrer Konstruktion für die Sequenz 0110101011 validiert. Eine Waveform für diese Beispielsequenz könnte folgendermaßen aussehen:

[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/sysarch21/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	$a \,\&\, b$
0	0	1	$a \mid b$
0	1	0	$\langle a \rangle + \langle b \rangle$
1	1	0	$\langle a \rangle - \langle b \rangle$
1	1	1	$0^{31}(\langle a \rangle < \langle b \rangle \; ? \; 1 : 0)$

Hauptteil

Die Gerüstdateien für den Hauptteil des Projekts finden Sie in unserem CMS unter Materialien[12]. 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 2.1: Arithmetic Logic Unit — 5 Punkte

Implementieren Sie das Modul `ArithmeticLogicModul` im Datenpfad und vervollständigen Sie die zugehörigen Steuerbits in der Dekodiereinheit gemäß Tabelle 2. Der 1-Bit Ausgang `zero` ist stets genau dann 1, wenn das Ergebnis der ALU `result[31:0]` null (0^{32}) ist.

Aufgabe 2.2: 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 2.3: Verzweigungen — 3 Punkte

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

Aufgabe 2.4: 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 2.5: 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 2.6: 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 2.7: 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.

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Lucas Biehl

System Architecture SS 2021 – Project 1

Hardware Design with Verilog

Submission Modalities

The project starts on June 9, 2021 with the release of this description. It consists of two parts: a warm-up part and a main part. The warm-up part is for you to get acquainted with Verilog as a language and with the programs we use. We recommend that you start working on the project *as soon as possible*.

You may work on the project in *groups of two to three people*. If you have formed a group, please create a corresponding team on your personal page in our CMS until

Monday, June 14, 2021, 23:59.

One person per group must upload the solutions of *both* parts of the project together to our CMS system by

Monday, June 28, 2021, 23:59

Both parts will be included in the evaluation of the project. A total of 32 points (plus 6 extra points) can be achieved. Projects submitted late will be graded with **0 points**. Use a ZIP file or a gzip compressed tar archive (i.e., `*.tar.gz`) for your submission. The archive should immediately contain all Verilog files (i.e., do not use subfolders). Use the skeleton files we provide, which already contain the necessary Verilog module declarations. Do not modify these module declarations unless it is explicitly allowed. Also, do not modify the existing filenames. If not all of these requirements are met, your submission cannot be evaluated.

Use the two sanitizer testbenches from the CMS to check your solution for illegal changes. To do this, run the following command:

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

(analogously for the main part). If everything compiles without problems and warnings, the sanitizer did not detect a violation. If our test detects a violation, adjust your submission accordingly.

If you worked on the project in a group, add a `contributions.txt` file to the archive that briefly describes how each team member contributed to the implementation. We may grade the project with 0 points for individual members if they have not made a significant contribution.

*Notes: Any collaboration with people who are not part of your own group is **not** allowed. We will check all submissions for plagiarism (against submissions from other groups, as well as submissions from previous years). Submissions that were created by modifying another project, for example by changing variable names, are also considered to be plagiarized. Plagiarized submissions will be graded with **0 points** and will be reported to the examination board as a cheating attempt; this may lead to expulsion from the university..*

*If you have attended the system architecture course in the past and would like to use your previous submission as a basis for the current project: This is only allowed for parts that you implemented yourself; code written by other members of your previous team may **not** be reused. In this case, add a file `previousyear.txt` to the submission that describes exactly which parts have been reused. Note, however, that the current project is not identical to projects from previous years. Submissions that only solve an old project will be graded with **0 points**.*

Tools and Documentation

For the synthesis and simulation of Verilog code, we will use *Icarus Verilog*[7], and for viewing generated waveforms, we will use *gtkwave*[8]. Both programs are available for Linux, macOS, and Windows. Detailed installation instructions can be found in the CMS under "Zusatzmaterial".

To synthesize a top-level module M, whose definition including all sub-modules is contained in the files `file1.v` to `filen.v`, run the following command on the command line

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

To start the simulation run the generated binary `sim`. Depending on the testbench, you will see your results on the command line, or you will find a generated waveform file, which you can view with `gtkwave`.

A good and very detailed introduction to Verilog with many examples is provided by Asic-World[9]. For information on using Icarus Verilog, see `http://iverilog.wikia.com/wiki/Getting_Started` and for GTKWave, see `http://gtkwave.sourceforge.net/gtkwave.pdf`.

For the main part of the project, you will need detailed information on the MIPS instruction set, in particular regarding the encoding of instructions. This information can be found on the official MIPS[10] website.

To run MIPS programs on your processor (e.g., for testbenches), you need the corresponding machine program. You can use the MARS simulator[11] to write MIPS assembler programs and to translate them into machine code. To do this, use `File->Dump Memory` and select `Hexadecimal text`. To be compatible with Verilog's `readmemh`, please use our customized version[12].

Questions and Problems

If you encounter any issues while working on the project that you cannot solve in your group, we will be happy to help you in the forum or during the office hours. In general, however, we expect that you have already done some work on your own (own tests, debugging outputs, online research, etc.) to solve the issue. Non-specific questions like "does the code compile?", "is this correct?", or "what needs to be done in problem X.Y?" will not be answered.

Warm-Up Part

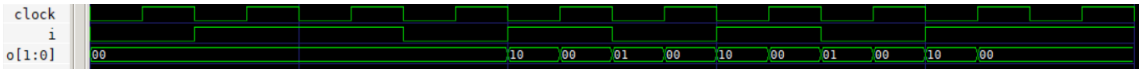
Start *as soon as possible* with this part of the project so that you have enough time left for the main part. The skeleton files for the project can be found in our CMS under materials[12].

Problem 1.1: Pattern Detection — 2 Points

You have already seen Mealy machines for detecting patterns on a previous assignment sheet. Here, we would like to detect the patterns 101 and 010 in a sequence of characters from the input alphabet $\{0, 1\}$. The output `o[1:0]` comes from the output alphabet $\{0, 1\} \times \{0, 1\}$. The left bit `o[1]`/right bit `o[0]` of the output shall be 1 if and only if the two previous inputs together with the current input form the pattern 101/pattern 010. For example, if the machine has detected the pattern 101, the output shall be 10. In the beginning (i.e., before sufficiently many input characters have been read), the output shall be 00.

Implement such a Mealy machine as a Verilog module `MealyPattern`.

Write a testbench `MealyPatternTestbench` that validates the correctness of your construction for the sequence 0110101011. A waveform for this example sequence could look as follows:



[7] `http://iverilog.icarus.com`
[8] `http://gtkwave.sourceforge.net`
[9] `http://www.asic-world.com/verilog/veritut.html`
[10] `https://www.mips.com/?do-download=the-mips32-instruction-set-v6-06`
[11] `http://courses.missouristate.edu/kenvollmar/mars/`
[12] `https://cms.sic.saarland/sysarch21/materials/`

Problem 1.2: Division Circuit — 5 Points

In the lectures, we have seen circuits for addition, subtraction and multiplication, but not for division. Integer division of two unsigned binary numbers can be implemented as a *sequential circuit*, based on the grade school division method. The division $\frac{\langle A \rangle}{\langle B \rangle}$ according to the school method can be expressed algorithmically as follows.

```
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
```

Exercise Compute $\frac{7}{3}$ using this algorithm.

Now design the corresponding sequential circuit. The division circuit has two 32-bit inputs A and B, a 1-bit input $start$, an input $clock$ and two 32-bit outputs Q and R, where Q is the quotient, and R is the remainder. 32 cycles after $start = 1$ at a rising clock edge, let $\langle Q \rangle$ be the quotient and $\langle R \rangle$ the remainder of the division $\frac{\langle A \rangle}{\langle B \rangle}$. If $start = 1$ occurs again during the computation of a division, the circuit aborts the current computation and starts over with the new operand values. In the following section, we provide some additional guidance.

Implement your sequential circuit as a Verilog module `division` and verify your design using testbenches.

Notes The sequential circuit shall perform *one* iteration of the loop per cycle between two consecutive rising clock edges, i.e. essentially one subtraction and one negativity test. Multiplication in hardware is expensive. Therefore, try to express the multiplication by cheaper shift operations.

Use three 32-bit wide registers to store the current state of the sequential circuit: The first register stores the current value of the remainder R. The second register stores the current value of the divisor B. The third register stores the remaining of the dividend A and the already computed bits of the quotient Q. Thus, before iteration i, the third register has the state

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

If at a rising edge of $clock$ the start signal is set ($start = 1$), we store the inputs A and B in the corresponding registers and start the computation. A computation takes exactly 32 cycles; after that, the correct result is available at the outputs until a new division starts. To achieve that *not in every* cycle an iteration is executed, but only in the first 32 cycles after the $start$, it can be helpful to use a counter.

Table 2: Control bits and operation of the arithmetic logic unit. The behavior for other assignments is undefined.

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

Main Part

The skeleton files for the main part of the project can be found in our CMS under materials[12]. For simulations, you can use the module `ProcessorTestbench`. We provide some test programs, which you need to uncomment in the testbench. Make sure to pass all these tests successfully.

You should additionally build your own testbenches to verify the correct operation of your circuits. Design suitable testbenches and think about the expected results *before* you start implementing them. Your testbenches for this part of the project will not be used for grading.

Problem: Single-Cycle MIPS Implementation — 0 Points

Familiarize yourself with the (almost complete) Verilog implementation of the single-cycle machine from the lecture. The machine so far supports the `addu`, `subu`, `and`, `or`, `sltu`, `lw`, `sw`, `addiu`, `beq`, and `j` instructions. Even though this task doesn't give you any points, take your time: once you have understood the structure of the datapath and the control unit, it will be much easier to work on the following problems.

Problem 2.1: Arithmetic Logic Unit — 5 Points

Implement the module `ArithmeticLogicModul` in the datapath and complete the corresponding control bits in the decoding unit according to Table 2. The 1-bit output `zero` is 1 if and only if the result of the ALU `result[31:0]` is zero (0^{32}).

Problem 2.2: Loading Constants — 4 Points

To load 32-bit constants, the two instructions `lui` and `ori` are very useful. Look up their encoding and operation in the MIPS instruction set documentation. Extend the datapath and the decoder to implement these instructions. Try to change the existing interface between the datapath and the decoder as minimally as possible.

Hint: It may be useful to already have the following exercises in mind when you modify the interface between the decoder and the datapath.

Problem 2.3: Branches — 3 Points

Implement the branch instruction `bne`. Try not to change the existing interface between the datapath and the decoder.

Problem 2.4: Multiplication — 7 Points

Implement the unsigned multiplication of the MIPS instruction set, more precisely the `multu` instruction. Think about in which module of the datapath the destination registers `HI` and `LO` should be placed, and what the logical state of a MIPS machine with multiplication is.

To process the result of a multiplication, the instructions `mflo` and `mfhi` are needed. Implement these two instructions. Try to change the existing interface between the datapath and the decoder as minimally as possible.

Problem 2.5: Function Calls — 6 Points

To support function calls efficiently, one needs a so-called *link register* to store the return address, which is needed to return to the caller at the end of a function call. The convention on MIPS machines is to use register 31. In assembler code, it is therefore also referred to as ra (*return address*).

Implement the `jal` and `jr` instructions for function calls and returns. Try to change the existing interface between the datapath and the decoder as minimally as possible.

Note: MIPS uses so-called *branch delay slots*. This means that the instruction located immediately after a branch or jump instruction is executed before the jump actually occurs. Therefore, the MIPS documentation lists `PC + 8` as the value of the link register. However, we do not consider delay slots for this project, and thus, the `jal` instruction shall write `PC + 4` to the link register.

Problem 2.6: Bonus: Division 1 — 2 Bonus Points

Implement the `divu` MIPS instruction using your division circuit from the warm-up part. Therefore, the division needs 32 cycles. During this time, the values of the `LO` and `HI` registers are considered to be unpredictable.

Use the `divu` instruction from page 171 of the MIPS documentation, which describes an older variant of the instruction that stores the result in `HI` and `LO`. More recent MIPS variants write the quotient directly to a general-purpose register.

Note: Use the registers `LO` and `HI` in a *clever* way, i.e., think about what they may store in the sequential circuit. *clever*[0.5cm] There is a dependency between the *read* operation of a `mflo/hi` instruction and the *write* operation of a preceding `divu` instruction. The division instruction needs several clocks to compute the correct result. In the meantime, the single-cycle machine is already processing the subsequent instructions. This makes the above read-write dependency problematic: it now depends on the number of instructions between `mflo/hi` and `divu` whether `mflo/hi` reads the correct result or an unpredictable value. Such a situation is therefore called a *hazard*. To avoid the above *hazard*, one should not execute `mflo` or `mfhi` for 32 cycles after a division. The programmer or the compiler must ensure that this convention, also called *Software Condition*, is observed.

Problem 2.7: Bonus: Division 2 — 4 Bonus Points

We would like to get rid of the *Software Condition* described above to make the programmer's life easier. Instead of resolving the *hazard* in software, one can also resolve such situations in hardware. For this, one uses a so-called *interlock*: if one detects that the current instruction accesses the `HI` or the `LO` register while a division is being executed, one "delays" the execution of the instruction.

Think about how such a "delay" can be implemented. Extend your division circuit to include an output `busy` that is 1 while a division is being executed. Implement an interlock mechanism for the situation described above.