SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Shrey Sharma

# System Architecture SS 2022 – Project 1

# Hardware Design with Verilog

## Submission Modalities

The project starts on June 8, 2022 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 13, 2022, 23:59.

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

Monday, June 27, 2022, 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 SanitizerWarmup *.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*[1], and for viewing generated waveforms, we will use *gtkwave*[2]. Both programs are available for Linux, macOS, and Windows. Detailed installation instructions can be found in the CMS under "Additional Material".

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[3]. 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[4] website.

To run MIPS programs on your processor (e.g., for testbenches), you need the corresponding machine program. You can use the MARS simulator[5] 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[6].

## 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.

# Warmup Part

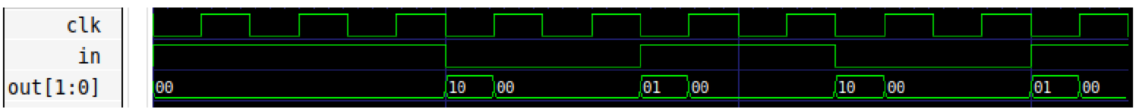
Start *as soon as possible* with this part of the project so that you have enough time left for the main part. The skeleton files for the project can be found in our CMS under materials[6].

## 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 110 and 001 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 110/pattern 001. For example, if the machine has detected the pattern 110, 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 1110011001. A waveform for this example sequence could look as follows:




[1] `http://iverilog.icarus.com`
[2] `http://gtkwave.sourceforge.net`
[3] `http://www.asic-world.com/verilog/veritut.html`
[4] `https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf`
[5] `http://courses.missouristate.edu/kenvollmar/mars/`
[6] `https://cms.sic.saarland/sysarch22/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 1: 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 | $0^{31}(\langle a\rangle < \langle b\rangle\ ?\ 1 : 0)$ |
| 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 | $a \,\&\, b$ |

# Main Part

The skeleton files for the main part of the project can be found in our CMS under materials[6]. 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 `ArithmeticLogicUnit` in the datapath and complete the corresponding control bits in the decoding unit according to Table 1. 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.

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.