ECE 2500 Project 2
Fall 2020

Total points: 100

Version: 1 (Any updates will be announced on Canvas.)

Deadline: Sunday Nov 15th at 11:59 PM

Late Policy: Projects have strict deadlines and have to be digitally submitted at 11:59 PM on the day they are due. If submitted before 1 AM, a 5% reduction in grade will apply and the project will still be accepted. If submitted by 11:59 PM of the day after the due date, a 20% reduction in grade will be applied and the project will still be accepted. No project will be accepted after that point without a letter from Dean's Office. In order for your projects to be accepted, you need to complete the submission process. Uploading files but not making them available to the instructor, submitting the wrong version, and other technical or non-technical issues do not make you eligible for a late submission.

Note 1: For this project, we will use Moss, a tool developed by Stanford University to detect similarities in assembly language or Verilog code structure. Moss cannot be defeated by changes to variable and function names, function ordering, formatting changes, and comments. Any strong similarities flagged by Moss will be carefully examined and possibly submitted to the Office of Undergraduate Academic Integrity.

Project Overview

Around 300 BC Euclid described an algorithm to compute the greatest common divisor of two numbers, which is the largest number that divides evenly into both numbers without leaving a remainder. However, Euclid's original algorithm requires use of the *mod* (remainder) function, which is an expensive operation. The binary GCD algorithm avoids the use of the *mod* operation and only requires subtraction, shift, and Boolean operations.

You can find an iterative C implementation of the binary GCD algorithm at https://en.wikipedia.org/wiki/Binary_GCD_algorithm. All numbers are assumed to be unsigned, and hence the shift operations can be logical shifts.

Part 1

Convert the iterative C algorithm to a MIPS assembly language function with name `binaryGCD`. The function should accept the *u* and *v* arguments in registers `$a0` and `$a1`, and return the result in register `$v0`. The function should use the `$t*` (caller-saved) registers beginning with `$t0` and hence does not need to save anything on the stack.

You should try to minimize the variety (rather than the number) of assembly language instructions used since you later have to implement these instructions. You should only use core instructions and avoid referencing memory (i.e. avoid using `lw` or `sw` instructions). Looking ahead to Part 3, the ALU Verilog code provided supports only the following instructions: `sll` (only by 1), `srl` (only by 1), `sub, subi, slt, or, ori, beq, bne, add, addi, and, andi,` and `lui`. These instructions allow arbitrary control flow, loading constants into registers, and performing simple arithmetic / Boolean / shift operations on unsigned numbers. The body of your `binaryGCD` function (except for the final "`jr $ra`" instruction) should try to limit itself to these instructions. The `main` function can use any instructions including pseudo-instructions since `main` code will not be executed in ModelSim. You should be able to create an unconditional branch (effectively a `jump`) using `beq`.

Test the `binaryGCD` algorithm with at least the following arguments:

1. gcd(12, 780) = 12
2. gcd(780, 12) = 12
3. gcd(2048, 16384) = 2048
4. gcd(500005199, 709) = 1
5. gcd(554400, 655200) = 50400
6. gcd(1048576, 131072) = 131072
7. gcd(0, 12345) = 12345

Perform each test by loading the arguments into `$a0` and `$a1`, calling `binaryGCD`, and then moving the i^{th} result (stored in `$v0`) to `$si`. This allows you and the GTA to quickly check the correctness of your `binaryGCD` function by inspecting the contents of `$s1` to `$s7` after all tests have been run. Ensure that your `binaryGCD` function has meaningful labels and comments that describe the function of each register.

Part 2

Ensure that your code works correctly by creating a single `testbench.s` file with a `main` function that calls your `binaryGCD` function once for each of the above test cases. The assembly instructions created should be preceded by the following assembler directives at the start of the file in order to have QtSpim assemble and load your instructions correctly:

```
.globl     main
.data
.text
```

```
main:
        < your assembly instructions to call binaryGCD  for each test case>

        ori   $v0, $0, 10
        syscall


binaryGCD:
        <your binaryGCD assembly instructions>

        jr    $ra
```

The `ori` and `syscall` instructions return from the `main` function to the run-time monitor. Be sure to always use the Qtspim `File -> Reinitialize and Load File` command after changing the assembler source code. Unlike Project 1, uncheck `Enable Delayed Branches` in the MIPS tab of the QtSpim `Preferences` to avoid the need to insert a delay slot instruction after a branch instruction; bad things will happen if you don't do this. It will be helpful to use the `Registers -> Decimal` formatting option. Run the code in QtSpim and check that `$s1` to `$s7` contain the expected values.

Part 3

Extend the Verilog-implemented datapath provided for the single-cycle MIPS architecture with the instructions needed to execute the `binaryGCD` calculation just once for the first test case (i.e. gcd(12, 780)). Unlike Part 2, `binaryGCD` will not be a function and hence the `$a0`, `$a1`, and `$v0` registers will not be used. Rather, the first instructions will load 12 into `$t0` (variable *u* in the C code) and 780 into `$t1` (variable *v* in the C code). The `shift` variable in the C code should use register `$t2`, and any additional temporary values should use registers `$t3`, `$t4`, ... The "`jr $ra`" instruction should be replaced with an infinite loop that branches to itself:

```
done: beq $t0, $t0, done
```

Using `$t0` in this instruction will conveniently cause ModelSim to display the result in the Waveform window. Following this guidance will make it easier for you and the GTA to validate your assembly code and Verilog.

The zipped `P2-start` folder includes the starting datapath. The instructions `sll`, `addi` and `lui` are already implemented within this datapath, however you do not necessarily have to use the `lui` instruction. All the modules needed to finish this project are given to you, although additional code is required in the `MIPS` and `MIPS_CONTROL` modules. The comments at the beginning of each file give you the description and important information about the module defined in that file. The modules for this project are written

in behavioral Verilog. You are not expected to understand how every module has been implemented. However, you need to recognize inputs, outputs and functionality of each module. Some of these modules are already instantiated in the top-level `MIPS` module, but some modules still need to be instantiated in the top-level module to complete the project.

In order to load ModelSim's instruction memory with your code that loads `$t0` and `$t1` with the gcd arguments, executes a single `binaryGCD` calculation, and then enters an infinite loop that reveals the result in `$t0`, you first need to type this code into a `program.s` file and have it assembled with QtSpim. It would also be a good idea to execute the code with QtSpim to ensure that the code produces the correct result. The assembled hex code should be cut and pasted from the QtSpim text window to a file named `program.txt`. Each line in the `program.txt` file should only have eight hexadecimal characters (without a leading "`0x`") and all spaces, address, source and comment fields should be removed. The first instruction should be copied from address 0x400024 (i.e. where you initialize $t0) and the last instruction should be the infinite loop. Count the total number of instructions in `program.txt` and replace the last argument in `instruction-mem.v`'s `$readmemh` system task with one less than this value (e.g. 39 if there were 40 instructions in `program.txt`). Your program cannot be longer than 128 words since the `IMEM` module only allocates a 128-word instruction array called `mem_array`.

You need to extend this initial datapath to support only the MIPS instructions used by your `binaryGCD` function. Support for load and store instructions need not be implemented if you do not reference data memory. Instruction implementation will require the instantiation of components and buses/wires in the `MIPS module`, and the generation of control signals in the `MIPS_CONTROL` module. The `ModelSim_refresher.pdf` document provides additional guidance for setting up a ModelSim project and running ModelSim.

What to turn in

1. Verilog modules: Create a folder called `Project2_PID` where `PID` is your PID. Put all the Verilog files associated with this part, *including the ones that you did not modify*, in this directory.
2. Test program: The code that includes a `main` function should be named `testbench.s`. The code that is loaded into ModelSim's instruction memory should be named `program.s`, with the assembled version named `program.txt`. Put all three of these files in the `Project2_PID` folder.

3. Report: Write a brief report that includes a description of your implementation and tests. Your report should be titled `Project2_report_PID` and included in the `Project2_PID` folder. Your report should include the following sections:
    a. A list of the instructions that were implemented.
    b. A summary of the changes made to the Verilog code to implement these instructions along with any unresolved problems.
    c. Any bonus features (such as additional instructions) and their tests.
    d. One or more QtSpim screenshots showing the results of executing `testbench.s`, including the `$s1` to `$s7` results from the seven calls to `binaryGCD` displayed in decimal.
    e. One or more ModelSim waveform screenshots showing the initial state of the simulation just after loading the gcd arguments into `$t0` and `$t1`, as well as the state of the simulation after completing the gcd calculation. The screenshots should include relevant signals such as `the` program counter and instruction, as well as relevant registers such as `$t0` and `$t1` before the computation and `$t0` after the computation.

Upload a zipped version of the `Project2_PID` folder to Canvas before the deadline.

Grading Scheme

Component	Points
`program.s` completeness and readability	20
Verilog code additions and readability	35
Demonstrate correctness of `binaryGCD` with QtSpim	10
Demonstrate correctness of hardware with ModelSim	20
Report discussions	15