EECS150 Lab Lecture 4
Serial I/O and Git

Vincent Lee
Electrical Engineering and Computer Sciences
University of California, Berkeley
Lab Logistics

• We recommend you finish this lab on time as the project design review is due the following week

• You may/must do this lab in partners
  – Both partners must be present for checkoff

• There will be one more lab lecture on the project next week

• The project starts next week
  – A design review will be during lab sections next week
  – Read specifications and start early
More Logistics

• Git repositories have been set up – verify that you have access to one and have been added to a team named “students”

• If you haven’t sent me your Github information with your partner, send me as soon as you find a partner

• If you can’t find a partner but still want one, send the email with your account information and “WANT PARTNER” also in the subject and we will try and match you with someone
Serial Communication

• Serial communication channel = 1 data channel

• Composed of transmitter and receiver
  – Must agree that transmission will take place (handshake)
  – Must indicate start of transmission (start bit)
  – Must indicate end of transmission (stop bit)

• Why Serial?
  – Multiple parallel buses with changing values = lots of interference
  – Lots of bus capacitance for close parallel wires
Ready Valid Interface

• Follows producer and consumer model
  – Three lines: DataValid, DataReady, DataIn
  – Producer
    • Output: DataValid – Data at output is valid and ready for transmission
    • Input: DataReady – Indicates consumer is ready to consume data
    • Output: DataIn – Data to be transmitted
  – Consumer
    • Input: DataValid – Indicates producer is ready to supply data
    • Output: DataReady – Ready to receive data from transmitter
    • Input: DataIn – Data to be consumed
Ready Valid Procedure

• Handshake protocol between transmitter and receiver
  – Transmitter has data to transmit = outputs valid signal to receiver
  – Receiver is ready to receive data = outputs ready signal to transmitter
  – When both ready and valid asserted
    • Transmitter drives Data line and outputs start sequence + data + stop sequence
    • Receiver data ready goes low until end of transmission to indicate it is not ready
  – Data line is meaningless until handshake occurs and transmission starts
Example: FIFO

- FIFO is way of providing data across clock domains
- Data often produced in spikes -> mitigates problem of overwhelming consumer
- Ready-valid interface between both producer and consumer
Universal Asynchronous Receiver/Transmitter (UART)

- Contains both a transmitter and receiver module for communication
- Serial line generally does not operate at same frequency as system using it
  - Baudrate tells you how much the frequencies differ
  - Baudrate = symbols per second
- Relies on ready/valid interface to allow asynchronous operation
  - DataInReady, DataInValid, DataOutReady, DataOutValid
  - Start bit is 1'b0, stop bit is 1'b1
  - By default the data line is high
Transmission Protocol

• Serial line frame rate is given by \( \text{ClockFreq} / \text{BaudRate} \)
  – 10 total frames per byte of transmission
  – Uses start and stop bits to mark data payload
• Start and stop bit
  – Serial line held high until transmission ready
  – Transmitter pulls serial line low (start bit = 1’b0)
  – Sends 8 bits = 1 byte of data at baudrate
  – Last bit is (stop bit = 1’b1)
• You will implement this part
Receiving Protocol

• Monitor serial line until start bit detected (serial line pulled low)
• Sample approximately halfway through each transmission frame (avoid metastable problem)
• After 8 bits transmitted, get stop bit and go back to monitoring line
• Already written for you in the lab distribution
UART is Asynchronous

• No set times for when data is transmitted
  – Allows data to be processed as it is available
  – Works well if data is produced at rate much slower than clocks

• Transmission protocol is slower than receiver and transmitter clocks

• Data bit changes every \((\text{Clock Frequency} / \text{Baudrate})\) cycles
  – Note that it’s relative to START of the handshake
This Week’s Lab

• Objective:
  – Implement and test a UART which will be used for your final project
  – Get you more familiar with the development flow using all the tools we’ve introduced
  – Figure out how to use Git and Subversion Control

• Pre-lab requirements:
  – Understand the ready/valid handshake interface for the UART

• Check off requirements:
  – Demonstrate the working UART by showing that the echo program we provide you works
  – Check in your functional UART to Github
Testing Setup

• An Echotestbench.v testing framework has been setup for you
• Equivalent simulation is shown in figure
• Should echo keyboard inputs to screen output
• Set up also reflects how modules are connected after synthesis
Using screen

- Will use screen to interface with serial line
- Keyboard input will be transmitted to UART
- Echo setup will reflect keystrokes back to console
- `screen` will receive key strokes and display to console
- To kill `screen` -> Ctrl-A + Shift-K
- To reattach improperly closed `screen` -> `screen -x`
- Make sure to kill `screen` – only one `screen` can be attached per serial cable
Recommendations

• Take a look at the receiver module – that will give you an idea of the complementing module to your transmitter

• Use a FSM to help you organize the state/phases of the transmission (ex. IDLE, TX, DONE)

• Define your control signals or state signals based off of you FSM model

• Look at the Modelsim waveform and confirm that what you see is what you expected

• Finish this lab on time – the design review for the project is due the next week
Tips and Pitfalls

• Understand the ready/valid interface
  – In some cases UART may pass testbench but not be completely correct
  – Off by one clock cycle will cripple your UART in the project

• Don’t assume simulation is wrong and that your circuit may work in hardware and not simulation
  – This is very rare. The simulator is there for a reason...
  – Synthesizing over and over again will eat a lot of time
  – If you think this is the case, run a synthesis in the background but try and get the simulation to work while it’s synthesizing
Relevance to Final Project

• Keyboard is on remote desktop not FPGA board
• Need a way to communicate with keyboard input to issue commands
• Processor will communicate with lab machine through UART
• UART communication protocol taken care of in BIOS software using memory mapped I/O
• Eliminates need to build keyboard peripheral interface on FPGA (for now...)
Subversion Control

- Version control is determined by “tree” of change sets
- Each change set holds changes with respect to previous commit node (points to previous node)
- **Local repository** is what you work on
- **Remote repository** is what’s on the server
- Usually want to push only working changes to the remote repository
  - Always want to have a working copy somewhere
Commit Structure

- **master** branch tag updates itself – main branch
- **HEAD** tag will point to most recent commit on remote repository
- Commits add nodes
- Reset will delete nodes
Github

• Will house your project’s remote repository

• You are STRONGLY encouraged to use subversion control
  – It is an industry standard
  – Keeps you organized and makes you more efficient
  – If your partner destroys your project you can revert it back to a working change set
  – Doing manual version control is painful and inefficient – there’s already enough to do

• Commit FREQUENTLY... it’s for your own good
Github Repository View
Project Repository Setup

- Each team will have their own remote repository set up for them
- Staff will provide a repository with skeletal files
  - Refer to project specification on how to pull files
  - Will be updated each checkpoint and if we add fixes to the skeleton
    (we reserve the right to add fixes)
  - Pulling from the staff directory should not break existing functionality across checkpoints
  - If something breaks it’s probably something you thought was working but actually wasn’t from a previous checkpoint
Repository Access Commands

- **git init**
  - Initializes an empty repository in the directory you are in

- **git clone <repository@location>**
  - Clones a repository from a remote server
  - Must have put RSA associated with your account put on Github (see lab manual)

- **git pull**
  - Fetches changes from your repository
  - Do this before you push your changes so that you are up to date

- **git push**
  - Pushes your changes to the repository

- **git push --tags**
  - Pushes your tags to the remote repository
Local Commit and Tag

• `git commit -am <message>`
  - Commits all tracked and changed files
  - Attaches `<message>` to change set description – good way to document change set
  - Do this when you reach a “checkpoint” in your coding
  - If you break your project later, you can revert to this point
  - Can also just commit one file

• `git tag -a <version> -m <tag name>`
  - Adds tag to current change set
  - Ex. `git tag -am v1.0 “Checkpoint 1 Done”`
  - Push tags to remote repository with `git push --tags`
Adding, Removing, and Ignoring Files

- **git add <file name>**
  - All files must be added before git will recognize them to be put on the remote repository
  - Do this when you have new important modules or test code you want to keep and push to the remote repository

- **git rm -f <file name>**
  - Removes a file from the repository
  - Will tell the local repository to delete file in next change set

- `.ignore`
  - Specifies which files or directories to ignore when producing change set
  - We pre-configure it to ignore files that get changed frequently like the build file
Status and Diff

• `git status`
  – Prints the status of your local repository and its files
  – Will show what files have not been added to the repository or haven’t been committed
  – You must commit and PUSH your changes in order for it to show up on the repository

• `git diff`
  – Shows which files have been modified on your local repository with respect to last commit
Change Set Merge Conflicts

• Occurs when you try and pull changes
  – Your partner modified a file you modified and got to Github first
  – Git will print you a nice merge conflict error message
  – Must resolve merge manually and commit again before you can push
• Git will mark the conflicting lines of code in the file(s) – your job to delete the mess it made and fix it
• Once you fix the conflicts, commit again and then push your changes
• Avoid modifying files that your partner is modifying
Branching

- `git branch <branchname>`
  - Creates a branch named `<branchname>`
  - Switch to branch by executing `git checkout <branchname>`
  - When you want to work on a feature but not destroy your working code
  - Allows you to keep a working main branch while you fiddle with your experimental branch or feature
  - Commit before you branch

- `git branch`
  - Display what branches are available and shows what branch you are on

- `git push origin <branchname>`
  - Pushes your branch to the remote repository so your partner can check it out

- `git branch -d <branchname>`
  - Kills and deletes a branch
  - Only do this if you are sure you want to
Checking Out Branches

- `git checkout <branch name>`
  - Switches your local repository to the specified branch
  - Branch must be created before it can be checked out
  - By default, you are on a branch called `master`

- `git checkout <commit hash>`
  - Checks out the commit with the specified SHA number in your commit structure
Merging Branches

• **git merge <branchname>**
  - When you want to add your changes on your branch to the master branch
  - Make sure you are **on** the branch you want to move the target branch to
    • Ex. If you want to merge branch new_feature to master, make sure you are on the master branch
  - Make sure to kill the branch if you done with it
  - Make sure you have most recent version of the branch if you and your partner are both working on it
General Advice

• If you don’t know what something does, DON’T DO IT
• Reclone the repository only in the most DIRE of circumstances
• Always push a WORKING copy to the master branch
• Avoid modifying a file your partner is working on – avoids merge conflicts
• Make sure your changes are pushed to the remote repository
• Use tags to mark milestones in project completion – allows you to find working change sets easier
• Use descriptive messages when adding commits – it will help in 2 or 3 weeks
• Communicate with your partner. Seriously this is important…
Questions and Answers

Q: Do I have to work with a partner on this lab?
A: No.

Q: Do I have to use Git and/or subversion control for the project?
A: No and yes. If you really want to you can manually subversion control your code. It just might take you several hours more. We do require each checkpoint to be turned in through Github and tagged though.
Questions and Answers

Q: My partner <name withheld> is dead weight and I’ve been doing everything myself. Can I dump my partner later in the project?

A: Yes. But it must be cleared and reviewed by a TA beforehand. It is better to resolve these issues sooner rather than later in the project. We highly discourage ditching your project partner but in the event that it does become a problem come talk to one of us in private. Projects that are completed individually will still be graded the same way as projects submitted by partners.
Questions and Answers

Q: I made a bad commit and it somehow got pushed to the repository. How do I get rid of it?
A: There is a Git reset command that can wipe out a commit but it’s tricky. You shouldn’t have pushed a broken change set to the remote repository in the first place.
Questions and Answers

Q: My repository blew up, has 10 different branches that don’t make any sense, and won’t commit anymore. Will the TA fix it for me?

A: No. We will not modify your Git repository at any time in this project. It is your responsibility to keep your repository clean and organized.

Q: Will the TAs answer Piazza at 2AM the day the project is due? How about 4AM? 6AM?

A: No absolutely not. It is your responsibility to budget your time and raise these questions before you finally realize you need to pull an all nighter. At this point your peers are your most valuable resource.
Next Week’s Lab (JK…)

• Next week’s lab lecture: Final Project
  – Pipelined Processor Implementation
  – Stalling the Processor

• Next week’s lab: No Lab – Begin Final Project and have Design Review
  – Prepare design review for next week’s lab section
  – Design review document and schematic are due the week after next
  – Begin implementing code – START EARLY, THIS IS THE HARDEST CHECKPOINT
  – Be prepared to be warm and cuddly with your project partner… or not…
Questions, comments, or concerns?