Notes and Tips for CMPEN331 Final Project

FINAL PROJECT: FORWARDING & FPGA IMPLEMENTATION

SLIDES COMPILED BY ANDREW JONES

Welcome, Students!

- If you are a student of CMPEN331 going through these slides, welcome!
- If you haven’t already, **please go watch the video.** These slides are meant to supplement the video, which contains information that may not be present in these slides.
- Feel free to use these slides as reference as you work on your Final Project.

Some Starting Notes

- As always, start early!
  - The Final Project is longer than the labs and about the same difficulty
  - **LATE SUBMISSIONS WILL NOT BE ACCEPTED FOR THE FINAL PROJECT AND WILL RECEIVE A ZERO**

  - This video assumes you have completed Lab 5
  - If you have not already completed Lab 5, You need to do so before starting the Final Project

  - The Bonus for the final project will be detailed in a separate video
  - However, you should watch this video first as the final project bonus video builds off of this video

What exactly are we doing in labs 3 through 5 + final project?

Labs 3 through 5, along with the final project, consists of implementing different stages/features of our five stage MIPS pipeline. The main structure of the five stages will be implemented through the labs, and the final polish and functionality will be implemented in the final project. There is also a bonus for the final project that implements additional instructions.

The bonus on final project is extra credit, I highly recommend that all of you try the bonus if you have time.

- Lab 3: Stages 1 and 2 (Instruction Fetch, Instruction Decode)
- Lab 4: Stages 3 and 4 (Execution, Memory Access)
- Lab 5: Stage 5 (Writeback)
- Final Project: Forwarding Behavior
- Final Project Bonus: Control Transfer Instruction Implementation (branch/jump instructions).

Overview of Final Project

FORWARDING

- Values that are available in the pipeline (i.e. not yet written to the regfile) and needed in the ID stage can be forwarded to reduce stalls and improve performance.
- This is done with two 4:1 multiplexers and some extra logic in the Control Unit
- Stalls will still be needed in some cases, which will be done with some more logic within the Control Unit

IMPLEMENTATION

- Once we’ve completed our processor, we will generate an implementation that can be written to an FPGA
- We first need to prepare our design for this (mainly accommodating I/O limits within the FPGA)
- Note: This step is not needed if you plan on doing the bonus

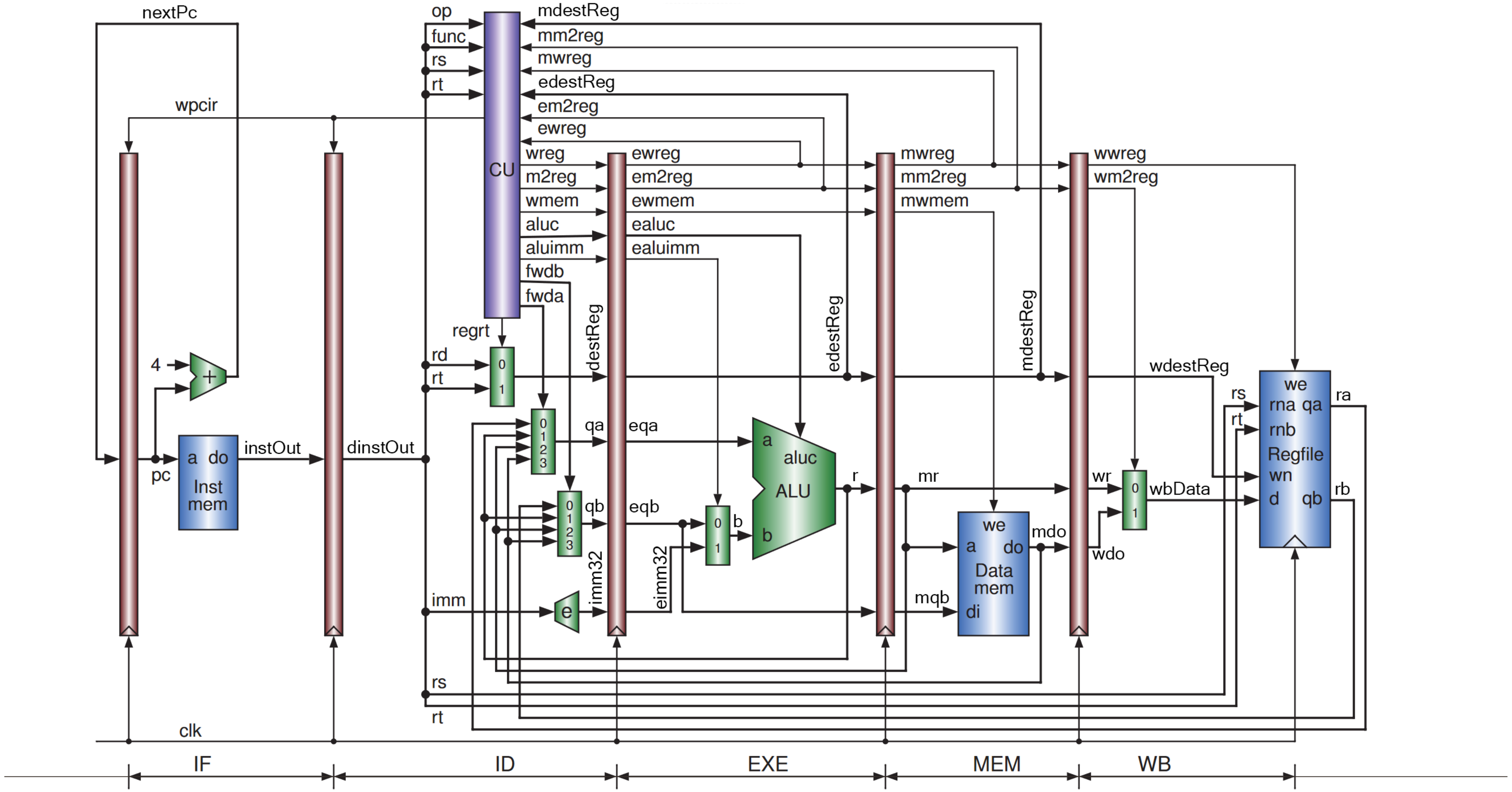
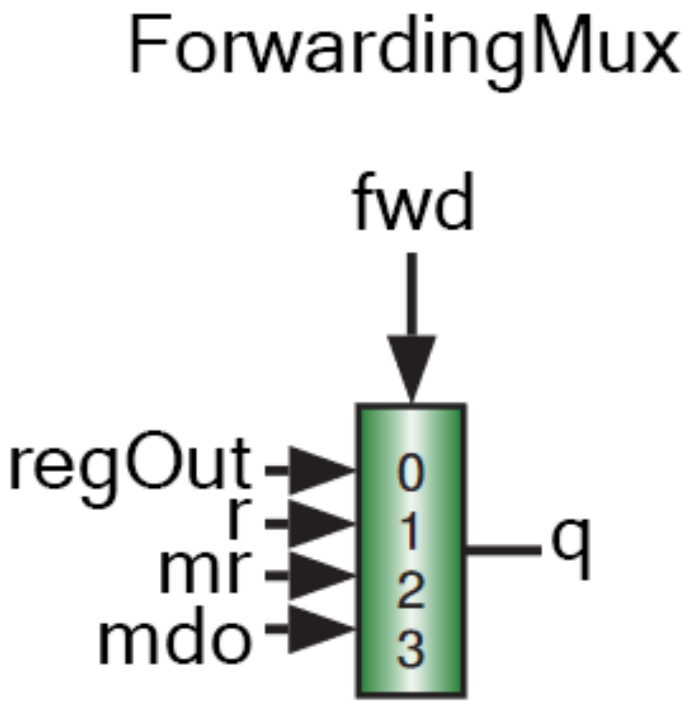


Diagram for Final Project



Forwarding Mux

Inputs:

- regOut [32 bits]
- r [32 bits]
- mr [32 bits]
- mdo [32 bits]
- fwd [2 bits]

Output:

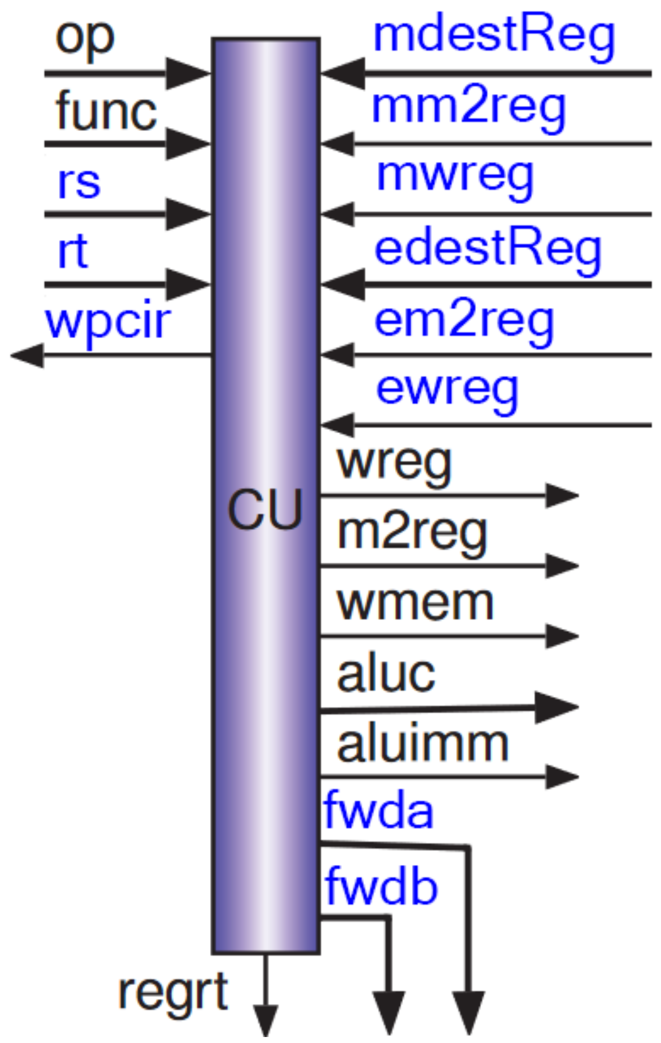
- q [32 bits]

Functionality: on any signal change

- If fwd is 0, q is set to value of regOut
- If fwd is 1, q is set to value of r
- If fwd is 2, q is set to value of mr
- If fwd is 3, q is set to value of mdo

Updated Control Unit

new signals in blue



Updated Control Unit

New Inputs:

- rs [5 bits]
- rt [5 bits]
- mdestReg [5 bits]
- mm2reg [1 bit]
- mwreg [1 bit]
- edestReg [5 bits]
- em2reg [1 bit]
- ewreg [1 bit]

New Outputs:

- fwda [2 bits]
- fwdb [2 bits]
- wpcir [1 bit]

New Internal Registers:

- regUsage [1 bit]

New Functionality will be described on next slide

A bit more on the new logic in Control Unit

- New Instructions:
  - We will be adding the following instructions to our control unit:
    - SUB
    - AND
    - OR
    - XOR
  - These instructions will all use the same control signals as ADD with the exception of the aluc signal, which will vary between these instructions
    - There are no set values for what the aluc should be set to for each operation, use whatever works best/makes the most sense to you
- Stalling:
  - We will need to stall if we have a load word instruction that is immediately followed by an instruction that uses the value being loaded by the load word instruction.
  - wpcir is our stall signal. It is active-low, meaning the pipeline registers stall if wpcir is 0 (rather than stalling if wpcir is 1).
  - New internal register regusage: 1-bit wide, set to 1 if rs or rt is being read
    - Unless you're doing the bonus, this signal will be set to 1 for all instructions
  - pseudocode for our wpcir signal (this goes outside of your case statement):

```
if (
           exe stage writes to regfile and
           exe stage writes from memory to register and
           exe stage destination register is not 0 and
           regusage is 1 and
           (exe stage destination register equals either rs or rt)
    )
begin
    wreg is set to 0
    wmem is set to 0
    wpcir is set to 0
end else begin
    wpcir is set to 1
    // don't set wreg and wmem here
end
```

A bit more on the new logic in Control Unit

- Forwarding:
  - We need to forward data if we have an instruction that uses a value that is in the pipeline but not yet written back into the register file
  - We could use stalling to handle this, but forwarding allows us to not stall and execute instructions faster at the cost of more hardware
  - Control Signals for forwarding will be based on the following pseudocode (this goes outside of your case statement):

```
if (
          exe stage writes to regfile and
          exe stage destination register is not zero and
          exe stage destination register equals rs and
          exe stage memory to register is zero
) fwda is set to 1
else if (
          mem stage writes to regfile and
          mem stage destination register is not zero and
          mem stage destination register equals rs and
          mem stage memory to register is zero
) fwda is set to 2
else if (
          mem stage writes to regfile and
          mem stage destination register is not zero and
          mem stage destination register equals rs and
          mem stage memory to register is one
) fwda is set to 3
else fwda is set to 0

repeat above for fwdb, but replace every “rs” with “rt”
```

Changes to other modules from Lab 5

- ALU
  - Your ALU should be modified to support the new aluc signals/operations. This should be trivial.
- Program Counter and IF/ID Pipeline Register
  - These modules will have a new 1-bit input: wpcir
  - This input simply is a write-enable. If it is zero, writes to these modules do not occur. This change should be trivial.
- Instruction Memory
  - The five instructions in instruction memory will be replaced by the following instructions:
    - `add $3, $1, $2`
    - `sub $4, $9, $3`
    - `or  $5, $3, $9`
    - `xor $6, $3, $9`
    - `and $7, $3, $9`
  - These will replace the load word instructions along with the one add instruction

Changes to other modules from Lab 5

- Datapath
  - Your Datapath needs to initialize the 2 new forwarding multiplexers and connect them together
  - You also need to rewire inputs and outputs from the control unit, along with the connected modules
  - You will also need to rewire the connections from the regfile
  - **This is the part students make the most mistakes on, so take your time and double-check every wire you're changing here. Use the diagram as a reference!**
- Testbench
  - Testbench will have no changes, use the same testbench you used for lab 5

- If you're doing the bonus for the project, you can go ahead and skip to the bonus video now, the rest of the information in this video is irrelevant to you.

Changes to other modules from Lab 5

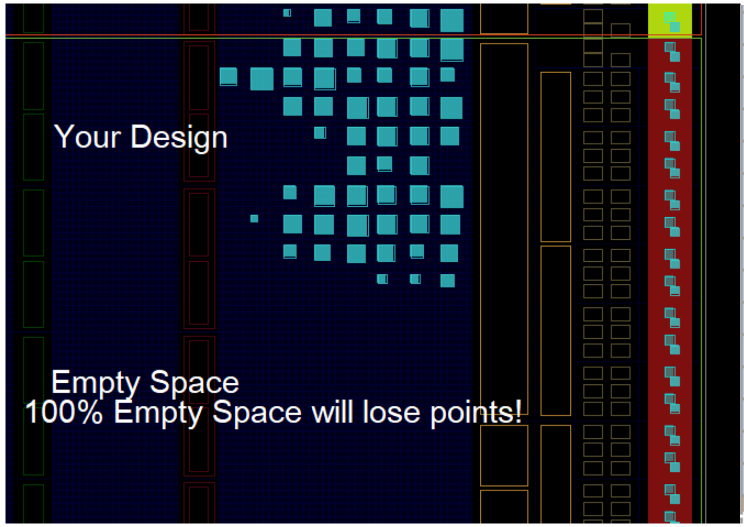
- Regfile (non-bonus):
  - If you are doing the bonus, don’t do any of the following and keep initializing all registers to zeros
  - If you are not doing the bonus, then instead of initializing all registers to zero, you will initialize the first 11 registers with the following hexadecimal values (You do not need to initialize the rest of the registers in the regfile):
    - 00000000
    - A00000AA
    - 10000011
    - 20000022
    - 30000033
    - 40000044
    - 50000055
    - 60000066
    - 70000077
    - 80000088
    - 90000099

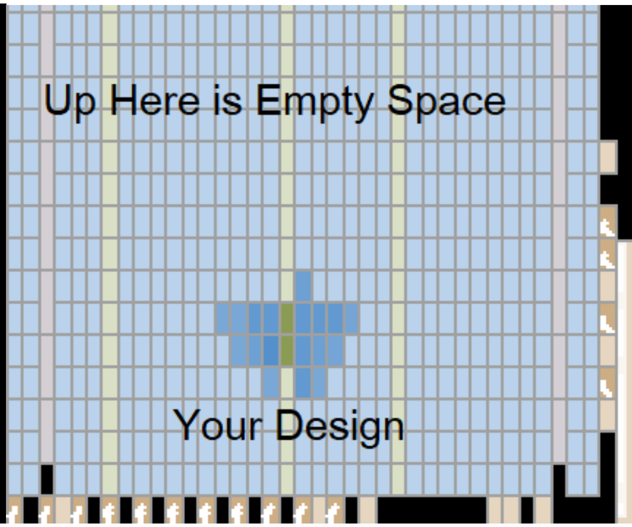
Implementation

- We’re now ready to implement our design
- **Reminder that you don’t need to do any of this if you’re doing the bonus, synthesis will suffice**
- However, our datapath module has 300+ connections (inputs and outputs), and our FPGA in Vivado only supports 100. Implementation will fail every time in this case.
- To solve this, we’ll use a top module that’ll only connect the pins we care about.
- The top module will have the following inputs and outputs (97 pins total):
  - clk [1 bit input]
  - eqa [32 bit output]
  - eqb [32 bit output]
  - mr [32 bit output]
- The top module will then instantiate the datapath similarly to the testbench, except we won’t connect anything other than clk, eqa, eqb, and mr:
  - `Datapath datapath(.clk(clk), .eqa(eqa), .eqb(eqb), .mr(mr));`
- This top module won’t do anything else. Make sure to set this module as your top module.
  - To do this, right click the module in the heirarchy and select “Set as Top”
    - in Quartus this is “Set as Top level Entity”
- DONT ADD THIS TO THE TESTBENCH!!!!! Testbench should be unchanged from Lab 5.

Implementation

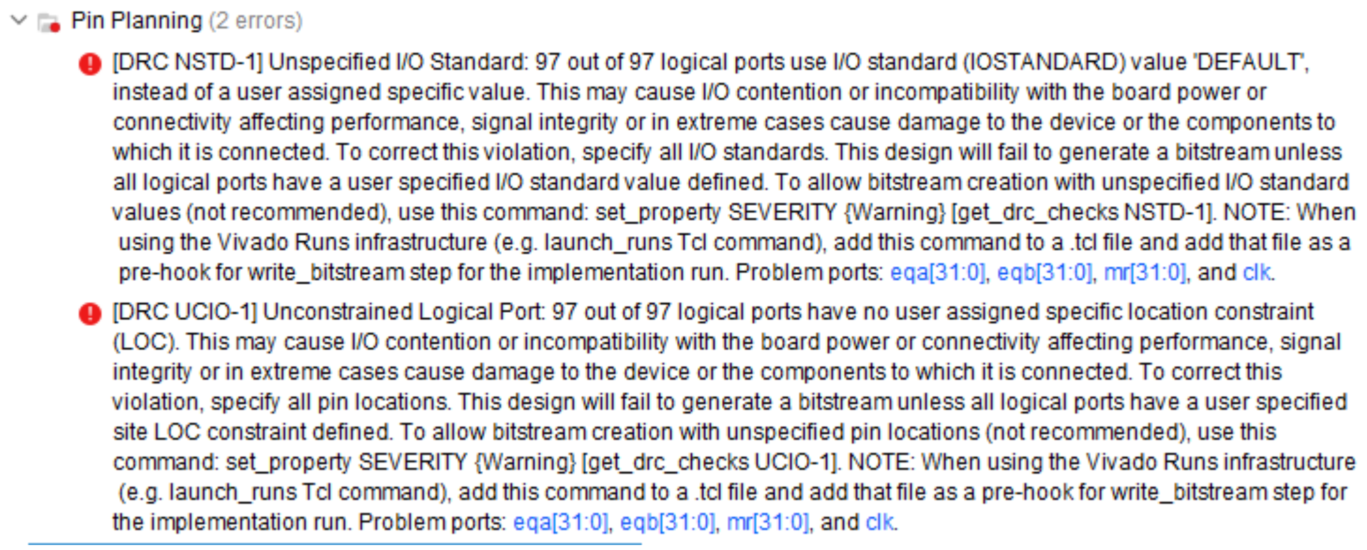
- Now that our top module is done, we can run implementation and generate the bitstream.
  - In Vivado, on the leftmost pane (Flow Navigator) click “generate bitstream”
  - In Quartus, you have a .sof file instead of a bitstream file and generation of the bitstream happens automatically when you click “start compilation”
- You can then use the IO planning and Floor Planning generated by Implementation
  - NOTE: the Floor Planning must not be empty! **Empty Floor Plannings will lose points!**
  - Implementation must also be of the whole design, not just one module. **This will also lose points!**
  - Bitstream or .sof files will be in the project directory. No need to move this elsewhere, just make sure to include all of the project directory. **Not including the project directory will result in a ZERO, no exceptions!!!**





Vivado Workaround

- One error students commonly get in Vivado when generating the bitstream is the following:



Pin Planning (2 errors)

[DRC NSTD-1] Unspecified I/O Standard: 97 out of 97 logical ports use I/O standard (IOSTANDARD) value 'DEFAULT', instead of a user assigned specific value. This may cause I/O contention or incompatibility with the board power or connectivity affecting performance, signal integrity or in extreme cases cause damage to the device or the components to which it is connected. To correct this violation, specify all I/O standards. This design will fail to generate a bitstream unless all logical ports have a user specified I/O standard value defined. To allow bitstream creation with unspecified I/O standard values (not recommended), use this command: set_property SEVERITY {Warning} [get_drc_checks NSTD-1]. NOTE: When using the Vivado Runs infrastructure (e.g. launch_runs Tcl command), add this command to a .tcl file and add that file as a pre-hook for write_bitstream step for the implementation run. Problem ports: eqa[31:0], eqb[31:0], mr[31:0], and clk.

[DRC UCIO-1] Unconstrained Logical Port: 97 out of 97 logical ports have no user assigned specific location constraint (LOC). This may cause I/O contention or incompatibility with the board power or connectivity affecting performance, signal integrity or in extreme cases cause damage to the device or the components to which it is connected. To correct this violation, specify all pin locations. This design will fail to generate a bitstream unless all logical ports have a user specified site LOC constraint defined. To allow bitstream creation with unspecified pin locations (not recommended), use this command: set_property SEVERITY {Warning} [get_drc_checks UCIO-1]. NOTE: When using the Vivado Runs infrastructure (e.g. launch_runs Tcl command), add this command to a .tcl file and add that file as a pre-hook for write_bitstream step for the implementation run. Problem ports: eqa[31:0], eqb[31:0], mr[31:0], and clk.

- To fix this, add a constraints.xdc file and add the following to it:

```
set_property SEVERITY {Warning} [get_drc_checks NSTD-1]

set_property SEVERITY {Warning} [get_drc_checks UCIO-1]
```

Final Remarks and Tips

- **I HIGHLY recommend putting different modules into different files.**
  - Yes, the lab doc tips say that you should put all non-simulation modules into a single file, but I've found that students who do use different files for modules spend less time debugging, and have an easier time finding modules when they need to.
  - Accessing the modules may require an extra menu, but from my experience it's much faster than scrolling up and down all of your code looking for the correct module.
- **Format your code.**
  - Proper indentation, formatting, and commenting of your code will make your life much, much easier when you enter the debugging stage of the lab.
  - Students who don't format and comment their code tend to spend a lot more time debugging than students who do format and comment.
  - How you format and comment your code is up to you, do what works best for you.
  - Remember that you will have to look at the code you write while debugging, not to mention the future labs and final project, so make sure it is easy for you to read and understand what's going on in your code.
- **Give everything names that make sense**
  - This fits in with the Format your code section.
  - The names of your modules, wires, and registers should make it fairly clear what they are used for or what they are trying to represent.
  - Naming registers reg1, reg2, etc. is ambiguous and will make debugging very difficult.
  - How you name your modules is completely up to you (with the exception of the testbench), but I recommend using the names within the provided diagrams.
    - Remember that with the testbench, your wires should very clearly represent what each signal is.
  - I also recommend defining module names with their intended stage of the pipeline as a prefix
    - For example, I would define the program counter module as "IF_ProgramCounter" and the regrt multiplexer as "ID_RegrtMux" with IF and ID corresponding to the correct stage of the pipeline.
  - Again, how you name everything is up to you, do what works best for you.

Final Remarks and Tips

- **Start Early!!!**
  - I know I put this tip at the beginning, but I cannot stress enough how important it is to give yourself enough time to complete the labs.
  - I recommend at minimum starting a week in advance of the due date, this will give you plenty of time to write your code, debug it, and access office hours.
- **Utilize Office Hours**
  - The office hours of TA’s and LA’s are provided on the syllabus.
  - TA’s and LA’s will gladly help with all aspects of the lab.
  - Note: Before asking TA’s and LA’s to debug code, please try and debug it yourself (remember, Follow The Data!)
  - If you can’t access office hours for whatever reason, you can still email TA’s and LA’s through Canvas or their PSU email and they will help you through there
    - TA and LA emails are also provided on the syllabus.
    - Please provide lots of detail about what’s causing you trouble in the lab, screenshots and Verilog code files help greatly with this!