# CDA 4150 Project #2[1]

Due: 11/8/05, 11:59pm

School of EECS, University of Central Florida, Orlando, FL 32816.

## Objective

You should convert your multi-cycle MIPS Verilog model from the previous lab into a pipelined MIPS model with full bypassing and a single branch delay slot. We expect this lab to be more challenging than the previous labs, so please start early. Remember to submit your lab electronically once you are finished.

We highly recommend you draw a "paper and pencil" block diagram showing the datapath of your pipelined processor (including the bypasses and the branch delay slot). The block diagram should include all pipeline registers, multiplexers, and major datapath elements, and should be labeled with the signal names from your Verilog model. Use vertical lines to divide the five pipeline stages and dashed vertical lines to show phases (half stages) where necessary.

You will almost certainly want to use the graphical debugging tool `virsims` for this lab. Please read this entire handout before beginning the lab, including the hints at the end of this handout.

## Lab Setup

Delete all large files from the previous lab that you do not absolutely need anymore (type `make clobber` in the main lab1 directory). Then go to your home directory (type `cd`). Then execute the following:

- `cd cda4150`
- `cp ~hgao/cda4150/lab2.tgz .`
- `tar -xzf lab2.tgz`
- `rm lab2.tgz`

Besides copying these files, you will need to make some other changes to the Verilog model before you begin the lab (some we have already made for you). These changes are outlined below.

This lab is not split into multiple parts. The entire lab will be graded out of 100 points.

### *mips.v*

In this file we have already changed the clock generation logic so that the posedge clock is at the beginning of the cycle, and we have extended the reset signal:

```
// Create a 50% duty cycle clock
initial begin
  CLK = 1;
  forever begin
    `PHASE
    CLK = 0;
    `PHASE
    CLK = 1;
  end
end

initial begin
  MRST = 1;
  #86 MRST = 0;
end
```

[1] The project was designed by Prof. Mark Heinrich for CDA4150 Fall 2004.

Without these changes, the register file will get corrupted during the boot code and cause problems later on.

### *rd.v*

We've added the following code to the end of the always block in `rd.v`:

```
// Special handling for syscall instructions:
// In the pipelined model, we need to stall for 2 cycles
// so that our syscall emulation routines will copy the right
// values from the register file
casex(I1[`op], I1[`function], I1[`rt])
  `SPECIAL, `SYSCALL, `dc5 : instIsSyscall = 1'b1;
  default: instIsSyscall = 1'b0;
endcase
```

In addition, `instIsSyscall` has been added to the port list for module `rd`, and declared as as both an output and a reg.

### *rf.v*

The following changes need only be made if you want to run sample before you add bypassing:

```
// Special handling for syscall instructions:
initial begin
  // Initializes register r0 with zeros
  RAM[0] = 32'b0;
  RAM[1] = 32'b0; // new code
  RAM[28] = 32'b0; // new code
  RAM[29] = 32'b0; // new code
End
```

After you implement bypassing, you should remove the new code (keep the line that sets r0 to 0!), as it will no longer be needed.

### *cpu.v*

We've added the following signal definitions under the `RD` signal definition section:

```
  wire decodeStall;
  wire instIsSyscall;
  reg rInstIsSyscall;
  reg rrInstIsSyscall;
```

And after the assignment to WriteCLK, the following code:

```
// ------------------------------------------------------------------
// This section is code that detects syscall instructions and stalls
// them in the decode stage for 2 cycles. This will become necessary
// in our pipelined model so that our syscall emulation routines
// will function properly. Do not change this code unless you have a
// *really* good reason to do so.
// ------------------------------------------------------------------
  always @(posedge CLK) begin
   rInstIsSyscall <= `TICK instIsSyscall;
   rrInstIsSyscall <= `TICK rInstIsSyscall;
  end
  assign decodeStall = (instIsSyscall | rInstIsSyscall) & ~rrInstIsSyscall;
```

`instIsSyscall` is also added to the port list where the `rd` module is instantiated.

# First Steps

Now you are ready to begin pipelining your processor. We have broken down the Verilog changes into two parts. The parts described below should all be done before you begin to add the bypassing logic.

## *Eliminating the State Register*

The current MIPS model is non-pipelined. As a result, there is a state register called `State` that changes its value with the processor's clock, enabling the **IF**, **RD**, **EX**, **MEM**, and **WB** stages sequentially.

- In a pipelined processor, all the stages are working concurrently. Eliminate the `State` and `nextState` registers. In fact, get rid of that entire always block. Another thing you will need to do is to change the PC-update circuitry in the `cpu.v` file.
- As you have seen, there is a need for pipeline registers between consecutive stages of the pipeline. We have already defined the **IR1**, **IR2**, **IR3**, and **IR4** instruction registers in your MIPS model. These registers transfer the instruction being executed between contiguous stages. You also need to pass the value of the `PC` between stages (for branch purposes). Call these registers `PC1`, `PC2`, `PC3`, and `PC4`. Note that these are not the only intra-stage registers that you need; there are additional data that must be passed between stages. Name these registers using your own convention with appropriate explanations in the README file.
- You are not required to deal with the problem of data dependencies in this section. To test your code, you must write small assembly files that do not involve data dependencies (" normal" programs will not work). In the `lab2/test` directory, you will find an assembly file called `sample.s` that executes a simple `printf` statement and then exits. The file is commented to help you create your own test assembly files. Make sure to include the names of the test files you create in the README file.

## *Syscalls*

The following information is **VERY IMPORTANT** for a working implementation:

A `syscall` (system call) instruction is the mechanism by which a program can ask for operating system services and is generated by C library calls like `printf`. We are not running an operating system, but in our model, we do properly handle all syscalls. When the `syscall` reaches the **EX** stage, it calls a special routine that copies the current content of the Verilog register file, performs the `syscall`, and then updates the Verilog register file with the results of the `syscall`. Since there may be instructions in the pipeline ahead of the `syscall` that need to modify the register file *before* the execution of the `syscall` instruction, it is necessary to stall the pipeline to wait for the instructions *ahead* of the `syscall` to leave the pipeline. Most of the subtleties for handling a `syscall` instruction have already been taken care of in the Lab Setup section above. However, you must implement the pipeline stalling logic yourself. There is a special flag called `decodeStall`. *Whenever* `decodeStall` *is high, you must make sure that:*

- The instruction register **IR2** is fed with a `NOP` instruction. This achieves the effect of stalling the pipe.
- The registers `PC1`, `PC2`, and `IR1` do **NOT** change their values. This prevents the pipeline from losing the instructions that must be executed after the `syscall` instruction.
- The `PC` register must **NOT** change.

However, when the `decodeStall` signal is low, the pipeline should function normally.

## *Delayed Branches*

The MIPS R3000 has one branch delay slot. The decision of whether a branch will be taken or not is made when the branch instruction is in the execution stage. At this same time:

- The delay slot instruction is in the decode stage, and

- A new instruction is being fetched from memory

This creates the following problem: if the branch is to be taken, then the instruction to be fetched next is NOT the one being fetched in the 2nd bullet above, but the one located at the branch target address. A possible solution to this problem is to create a second delay slot (i.e., two instructions after the branch will be executed). However, this is not the way the MIPS R3000 works. To maintain a *single* delay slot, your model must be changed so that the instruction fetch is not done at the rising edge of the clock, but rather at the *falling* edge of the clock. In this manner, the decision for branching can be made *before* the instruction fetch takes place, and thus the correct instruction can be fed into the pipeline. This effectively mimics what the MIPS R3000 does by delaying the instruction fetch by an extra half-phase as discussed in class.

So you must change the always block that updates the `PC` to be triggered off `negedge clk`, as well as modifying the update logic to remove the `State` dependence as described previously in this handout and to properly handle any stall conditions.

## Data Dependencies: Bypassing

To solve the problem of data dependencies between instructions inside of the pipeline, you must add bypass logic to your model. To create the bypass logic, you must add multiplexers in the **EX** stage and control logic that decides which input (the value from the register file, the **MEM** stage, or the **WB** stage) of the multiplexer is selected.

Make the necessary changes to the MIPS model to handle all possible situations where a bypass is needed. Make sure to add the control logic in a separate file called `bp.v`. Once you have implemented bypassing, you should once again be able to run "normal" MIPS programs.

## Extra Credit: Multiplication/Division (10 points each)

Since the multiply and divide instructions only write the `hi` and `lo` registers, you do not need to include the multiplier or divider in any bypassing logic. Your pipelined processor should ship multiplies and divides to the multiply/divide unit and keep operating the pipeline normally even though the multiply or divide may still be in progress. The only thing you need to do is ensure that any `mfhi` and `mflo` instruction stalls in the decode stage when the multiplier or divider is busy, and that you detect structural hazards on the multiplier and divider. For full credit on each part you must implement both the signed and unsigned versions of the operation (e.g. mult and multu, div and divu).

## Submitting Your Results

Submit your files electronically once you have finished. The procedure is the same as in the previous labs. In addition, you should begin by drawing the block diagram of the processor (with the bypassing multiplexers and pipeline registers).

If you have special instructions for the graders, you must inform us in the README file. To submit your files, make sure you are in the `lab2` directory and type

```
~hgao/cda4150/submit4150 lab2
```

Only one person of the group needs to submit. You may submit several times if you wish, but the newest submit will completely overwrite all previous ones.

Before you submit, make sure:

- to `make clobber` (points will be taken off if you do not do this)
- to delete unused files (or move them to another directory that is not a subdirectory of lab2)
- your code is `vcheck` clean
- your code compiles (you will receive zero points if it does not!)
- you only submit one CPU

- your code is commented
- you use CLK and not clk
- you remove (comment out) all debugging code
- your makefile does not automatically start a simulation
- you do not use any delays in your Verilog code other than the standard `TICK with flops

## Lab Hints

- Note that the boot code contains dependencies but nevertheless runs without bypassing. It will, however, initialize the registers with funny values when run without bypassing.
- If you want `gmake` to automatically make your assembly files, append their names (without extension) to the " IMAGES = …" line in the makefile.
- If the values in your registers are not correct at the time a syscall "executes", you will get an unimplemented syscall error.
- Jumps are handled like branches, i.e., they have a single delay slot, are part of the half-cycle trick, and need to get the correct inputs in the EX stage.
- `li` and `la` are pseudo instructions that the assembler automatically converts into one or more real MIPS instructions.
- Realizing that the model you start this lab with runs test/sample correctly, you might want to keep a copy around to compare against as you work through this lab.
- The bypassing logic should not match on register r0, even if a previous instruction writes to it.