EECS 150 Spring 2012 Checkpoint 1: Pipelined MIPS Processor

Prof. John Wawrzynek
TAs: James Parker, Daiwei Li, Shaoyi Cheng
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

Revision 1

1 Introduction

The first checkpoint in this project is designed to guide the development of a three-stage pipelined MIPS CPU that will be used as a base system in subsequent checkpoints. The MIPS processor will be implemented in Verilog by 1-2 person teams.

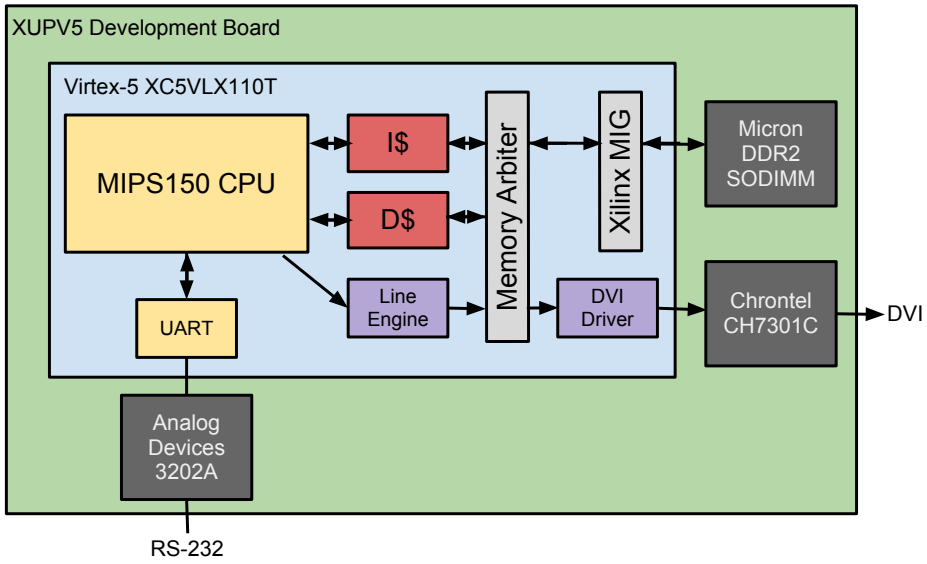

Figure 1: A high-level overview of the final system

The yellow blocks are the focus of the first checkpoint. Next, you will experiment with different cache implementations, shown in red. Finally, you will implement the purple blocks to enable 2-D accelerated graphics. The black blocks are physical chips on the board that your design will interact with.

2 MIPS 150 ISA

In the following tables, $R[\$x]$ indicates the register with address x, PC is the memory address of the instruction, $SEXT$ and $ZEXT$ are sign and zero extension, and H-, B- and $WMEM$ refer to byte, half, and word aligned memory accesses.

Table 1: RTL Specification

Mnemonic	**RTL Description**	**Notes**
LB	$R[\$rt] = SEXT(BMEM[(R[\$rs] + SEXT(imm))[31:0]])$	delayed
LH	$R[\$rt] = SEXT(HMEM[(R[\$rs] + SEXT(imm))[31:1]])$	delayed
LW	$R[\$rt] = WMEM[(R[\$rs] + SEXT(imm))[31:2]]$	delayed
LBU	$R[\$rt] = ZEXT(BMEM[(R[\$rs] + SEXT(imm))[31:0]])$	delayed
LHU	$R[\$rt] = ZEXT(HMEM[(R[\$rs] + SEXT(imm))[31:1]])$	delayed
SB	$BMEM[(R[\$rs] + SEXT(imm))[31:0]] = R[\$rt][7:0]$	
SH	$HMEM[(R[\$rs] + SEXT(imm))[31:1]] = R[\$rt][15:0]$	
SW	$WMEM[(R[\$rs] + SEXT(imm))[31:2]] = R[\$rt]$	
ADDIU	$R[\$rt] = R[\$rs] + SEXT(imm)$	
SLTI	$R[\$rt] = R[\$rs] < SEXT(imm)$	
SLTIU	$R[\$rt] = R[\$rs] < SEXT(imm)$	unsigned compare
ANDI	$R[\$rt] = R[\$rs] \wedge ZEXT(imm)$	
ORI	$R[\$rt] = R[\$rs] \vee ZEXT(imm)$	
XORI	$R[\$rt] = R[\$rs] \oplus ZEXT(imm)$	
LUI	$R[\$rt] = \{imm, 16'b0\}$	
SLL	$R[\$rd] = R[\$rt] << shamt$	
SRL	$R[\$rd] = R[\$rt] >> shamt$	
SRA	$R[\$rd] = R[\$rt] >>> shamt$	
SLLV	$R[\$rd] = R[\$rt] << R[\$rs]$	
SRLV	$R[\$rd] = R[\$rt] >> R[\$rs]$	
SRAV	$R[\$rd] = R[\$rt] >>> R[\$rs]$	
ADDU	$R[\$rd] = R[\$rs] + R[\$rt]$	
SUBU	$R[\$rd] = R[\$rs] - R[\$rt]$	
AND	$R[\$rd] = R[\$rs] \wedge R[\$rt]$	
OR	$R[\$rd] = R[\$rs] \vee R[\$rt]$	
XOR	$R[\$rd] = \overline{R[\$rs]} \oplus \overline{R[\$rt]}$	
NOR	$R[\$rd] = \overline{R[\$rs]} \wedge \overline{R[\$rt]}$	
SLT	$R[\$rd] = R[\$rs] < R[\$rt]$	
SLTU	$R[\$rd] = R[\$rs] < R[\$rt]$	unsigned compare
J	$PC = \{PC[31:28], target, 2'b0\}$	delayed
JAL	$R[31] = PC + 8; PC = \{PC[31:28], target, 2'b0\}$	delayed
JR	$PC = R[\$rs]$	delayed
JALR	$R[\$rd] = PC + 8; PC = R[\$rs]$	delayed
BEQ	$PC = PC + 4 + (R[\$rs] == R[\$rt] ? SEXT(imm) << 2 : 0)$	delayed
BNE	$PC = PC + 4 + (R[\$rs] != R[\$rt] ? SEXT(imm) << 2 : 0)$	delayed
BLEZ	$PC = PC + 4 + (R[\$rs] <= 0 ? SEXT(imm) << 2 : 0)$	delayed
BGTZ	$PC = PC + 4 + (R[\$rs] > 0 ? SEXT(imm) << 2 : 0)$	delayed
BLTZ	$PC = PC + 4 + (R[\$rs] < 0 ? SEXT(imm) << 2 : 0)$	delayed
BGEZ	$PC = PC + 4 + (R[\$rs] >= 0 ? SEXT(imm) << 2 : 0)$	delayed

Table 2: ISA Encoding

31 26	25 21	20 16	15 11	10 6	5 0	
opcode	rs	rt	rd	shamt	funct	R-type
opcode	rs	rt	immediate			I-type
opcode	target					J-type
Load and Store Instructions						
100000	base	dest	signed offset			LB rt, offset(rs)
100001	base	dest	signed offset			LH rt, offset(rs)
100011	base	dest	signed offset			LW rt, offset(rs)
100100	base	dest	signed offset			LBU rt, offset(rs)
100101	base	dest	signed offset			LHU rt, offset(rs)
101000	base	dest	signed offset			SB rt, offset(rs)
101001	base	dest	signed offset			SH rt, offset(rs)
101011	base	dest	signed offset			SW rt, offset(rs)
I-type Computational Instructions						
001001	src	dest	signed immediate			ADDIU rt, rs, signed-imm.
001010	src	dest	signed immediate			SLTI rt, rs, signed-imm.
001011	src	dest	signed immediate			SLTIU rt, rs, signed-imm.
001100	src	dest	zero-ext. immediate			ANDI rt, rs, zero-ext-imm.
001101	src	dest	zero-ext. immediate			ORI rt, rs, zero-ext-imm.
001110	src	dest	zero-ext. immediate			XORI rt, rs, zero-ext-imm.
001111	00000	dest	zero-ext. immediate			LUI rt, zero-ext-imm.
R-type Computational Instructions						
000000	00000	src	dest	shamt	000000	SLL rd, rt, shamt
000000	00000	src	dest	shamt	000010	SRL rd, rt, shamt
000000	00000	src	dest	shamt	000011	SRA rd, rt, shamt
000000	rshamt	src	dest	00000	000100	SLLV rd, rt, rs
000000	rshamt	src	dest	00000	000110	SRLV rd, rt, rs
000000	rshamt	src	dest	00000	000111	SRAV rd, rt, rs
000000	src1	src2	dest	00000	100001	ADDU rd, rs, rt
000000	src1	src2	dest	00000	100011	SUBU rd, rs, rt
000000	src1	src2	dest	00000	100100	AND rd, rs, rt
000000	src1	src2	dest	00000	100101	OR rd, rs, rt
000000	src1	src2	dest	00000	100110	XOR rd, rs, rt
000000	src1	src2	dest	00000	100111	NOR rd, rs, rt
000000	src1	src2	dest	00000	101010	SLT rd, rs, rt
000000	src1	src2	dest	00000	101011	SLTU rd, rs, rt
Jump and Branch Instructions						
000010	target					J target
000011	target					JAL target
000000	src	00000	00000	00000	001000	JR rs
000000	src	00000	dest	00000	001001	JALR rd, rs
000100	src1	src2	signed offset			BEQ rs, rt, offset
000101	src1	src2	signed offset			BNE rs, rt, offset
000110	src	00000	signed offset			BLEZ rs, offset
000111	src	00000	signed offset			BGTZ rs, offset
000001	src	00000	signed offset			BLTZ rs, offset
000001	src	00001	signed offset			BGEZ rs, offset

This is a subset of the full ISA that still allows for interesting programs while greatly simplifying the design. This subset omits floating point, coprocessor, trap, multiplication, division and a several other instructions that are of little utility for this project.

Note that the RTL specification marks some instructions as delayed. This ISA includes 1-cycle architected branch and load delay slots, which means the instruction following the branch or load is always executed, and must not depend on the outcome or data of the branch or load.

3 Pipelining

Your CPU must implement this instruction set using a 3 stage pipeline. The division of the datapath into three stages is left unspecified as it is an important design decision with significant performance implications. We recommend that you begin the design process by considering which elements of the datapath are synchronous and in what order they need to be placed. After determining the design blocks that require a clock edge, consider where to place asynchronous blocks to minimize the critical path.

4 Hazards

As you have learned in lecture, pipelines create hazards. The branch delay slot eliminates control hazards, but you will still need to resolve data hazards. You must resolve data hazards using forwarding. You may not stall the CPU to resolve hazards.

Additionally, you are restricted to using positive-edge triggered state elements in your design. The textbook uses a negative-edge writeback to the register file to eliminate one forwarding path, but this is not a good design decision in a three-stage CPU. Ask a TA if it is unclear why.

5 Memory Architecture

The datapath shown in DDCA has separate instruction and data memories. Although this is an intuitive representation, it does not let us modify instruction memory to run new programs. Your CPU, by the end of this checkpoint, will be able to receive MIPS binaries through a serial interface, store them into instruction memory, then jump to the downloaded program. To facilitate this, we will adopt a modified memory architecture, shown in Figure 2:

These block rams have been provided for you in `/hardware/src/imem_blk_ram` and `/hardware/src/dmem_blk_ram`. See Appendix A for details about initializing and using the block RAMs.

5.1 Address Space Partitioning

Your CPU will need to be able to access multiple sources for data as well as control the destination of store instructions. In order to do this, we will partition the address space into three main categories: data memory read and writes, instruction memory writes, and memory-mapped I/O. This will be encoded in the top nibble of the memory address generated in load and store operations, as shown in Table 3. In other words, the target device of a load or store instruction is dependent on the address. (Note: for this checkpoint, this does not apply for the PC. Instruction fetch is always from the instruction memory.)

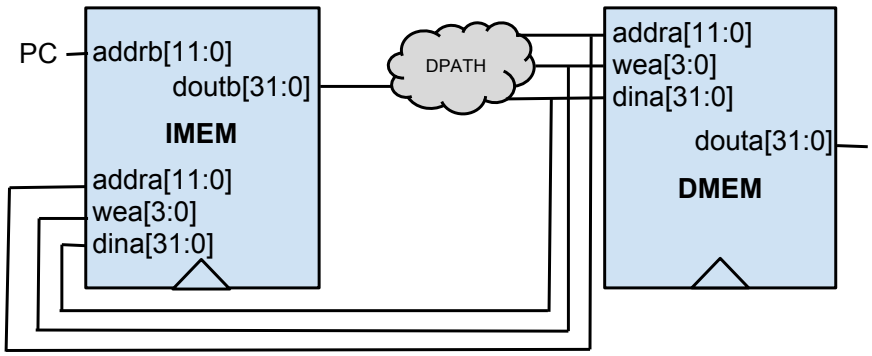

Figure 2: Initial Memory Architecture

Table 3: Memory Address Partitions

Address[31:28]	**Device**	**Access**
4'b0xx1	Data memory	Read/Write
4'b0x1x	Instruction Memory	Write Only
4'b1000	I/O	Read/Write

Each partition specified in Table 3 should be enabled only based on its associated bit in the address encoding. This allows operations to be applied to multiple devices simultaneously, which we will be used to maintain memory consistency between the data and instruction memory. For example, a store to an address beginning with 0x3 will write to both the instruction memory and data memory, while storing to addresses beginning with 0x2 or 0x1 will write to only the instruction or data memory, respectively. For details about the BIOS and how to run programs on your CPU, see Appendix D.

5.2 Memory Mapped I/O

At this stage in the project the only way to interact with your CPU is through the serial interface. The UART from Lab 5 accomplishes the low-level task of sending and receiving bits from the serial lines, but you will need a way for your CPU to send and receive bytes to and from the UART. To accomplish this, we will use memory-mapped I/O, a technique in which registers of I/O devices are assigned memory addresses. This enables load and store instructions to access the I/O devices as if they were memory. Table 4 shows the memory map for this stage of the project.

You will need to determine how to translate the memory map into the proper ready-valid handshake signals for the UART. Essentially, you will want to set the output valid/ready/data signals from the CPU based on the calculated load/store address, and mux in the appropriate valid/ready/data signals to be written back to the register file.

Table 4: I/O Memory Map

Address	Function	Access	Data Encoding
32'h80000000	UART transmitter control	Read	{31'b0, DataInReady}
32'h80000004	UART receiver control	Read	{31'b0, DataOutValid}
32'h80000008	UART transmitter data	Write	{24'b0, DataIn}
32'h8000000c	UART receiver data	Read	{24'b0, DataOut}

6 Testing

The design specified for this project is a complex system and debugging can be very difficult without tests that increase visibility of certain areas of the design. Although we will not require or grade testing efforts, we expect that teams utilizing the testing tools will be able to complete checkpoints faster. We strongly encourage that you follow the suggestions here for testing. A reasonable order in which to complete your testing is as follows:

1. Test that your modules work in isolation via Verilog testbenches
2. Test the entire CPU one instruction at a time with an assembly program
3. Test serial I/O - see the provided file `EchoTestbench.v`

6.1 Simulation

You learned how to write Verilog testbenches and simulate your Verilog modules in Lab 3. You should use what you learned to write testbenches for all of your sub-modules. As you design the modules that you will use in your CPU, you should be thinking about how you can write testbenches for these modules. For example, you may want to create a module that handles all of the branching logic. You could imagine testing this module in a way similar to how you tested the ALU.

6.2 Integration Testing

Once you are confident that the individual components of your processor are working in isolation, you will want to test the entire processor as a whole. The easiest way to do this is to write an assembly program that tests all of the instructions in your ISA. A skeleton is provided for you in `software/src/asmtest`. See Appendix C for details.
Once you have verified that all the instructions in the ISA are working correctly, you may also want to verify that the memory mapped I/O and instruction/data memory reading/writing work with a similar assembly program.

7 Checkoff

The checkoff for this specification is divided into two stages: block diagram/design and implementation. The second part will require significantly more time and effort than the first one. As such, completing the block diagram in time for the design review is crucial to your success in this project.

In non-checkoff weeks, we will require that your team meets with a TA during lab section to discuss project progress and any issues you encounter. For each progress meeting, we will have a rough guideline of expected results, though these won't be graded as the checkoffs are. The progress meetings are primarily intended to help you stay on track, solve problems before you become deadlocked, and give the TAs as much insight into your work as possible in the event you run into design problems.

7.1 Checkpoint 0.5: Block Diagram and Sub-modules

The first checkpoint requires a detailed block diagram of your datapath. The diagram should have a greater level of detail than the diagrams in DDCA. You may complete this electronically or by hand. If working by hand, we recommend working in pencil and combining several sheets of paper for a lager workspace. Though the textbook diagrams are a decent starting place, please remember that they use asynchronous-read memories and we will be using synchronous-read block RAMs (which you should recall from Lab 4).

Additionally, at this point you must have completely functional UART, ALU, ALU controller, and Register File modules (see C.1). **Checkpoint 0.5 is due in lab no later than 2:00 PM, Wednesday, February 29.**

7.2 Progress Meeting, Week 8

In labs, you probably found that you spent significantly more time debugging and verifying your design than actually writing Verilog. Though your skills are continually improving, this project involves a complex system and as such, bugs are inevitable. Design verification can take more than twice as long as writing the initial implementation. Given this, we recommend that you have completed your first stab at writing the Verilog and associated module testbenches for your processor by this check-in. Additionally, we would also like to see evidence of integration testing for your design. Make sure to meet with a TA in lab on either **Tuesday, March 6** or **Wednesday, March 7**.

7.3 Checkpoint 1: Base MIPS150 System

This checkpoint requires a fully functioning three stage MIPS CPU as described in this specification. Checkoff will consist of a demonstration of the BIOS functionality, storing a program over the serial interface, and successfully jumping to and executing the program. **Checkpoint 1 is due in lab no later than 2:00 PM, Wednesday, March 14.**

A Block RAMs

A.1 Initialization

Inside of `hardware/src/imem_blk_ram` and `hardware/src/dmem_blk_ram` there are three skeleton files:

- `*mem_blk_ram.xco`: This file contains configuration information used by coregen to build the memory. The only attribute you will need to change is the `coe_file`. To initialize the memories with a program for synthesis, set this field to point to the desired `.coe` file and re-generate the memories. This does not initialize the memory for simulation.
  Tip: copy the `.coe` file you want to use into the directory where the `.xco` file resides.
- `build`: Running `./build` generates the memory based on the configuration information. Run this if you change the parameters in the `.xco` file. You must run this if you decide to use a different `.coe` file.
- `clean`: Run `./clean` to delete the files created when you generate the memories. Do not run this from the GUI or any other directory!

The skeleton files contain two programs that you will likely want to initialize your memories with: `bios150v3` and `echo`. The bios is significantly more complicated, so while debugging, you may want to stick with `echo` until it works on the hardware.

In simulation, the memories are initialized with a `.mif` file. The software toolchain generates these for you; look at the `hardware/sim/test/*.do` files for examples of how to use these.

A.2 Endianness

The block RAMs have 4096 32-bit rows, as such, they accept 12 bit addresses (the block rams are **word-addressed**). However, the memory address that the processor computes is **byte addressed**. Thus, the bottom 14 bits of the addresses computed by the CPU are relevant for block RAM access. The top 12 are the word address (for indexing into the block RAM), and the bottom two are the byte offset.

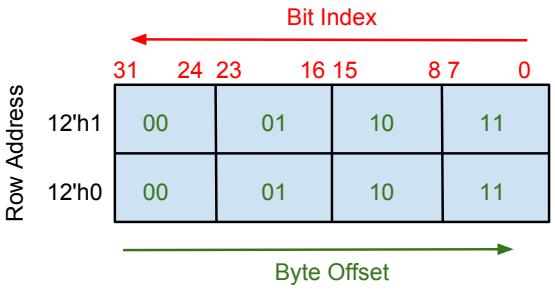

Figure 3: Block RAM organization

Figure 3 illustrates the 12-bit word addresses and the two bit byte offsets. Observe that the RAM is big-endian, i.e. the most significant byte is at offset 00. Since your block RAMs have 32-bit

rows, you can only read out data out of your block RAM 32-bits at a time. This is a problem when you want to execute a lh or lb instruction, as there is no way to indicate to the block RAM which 8 or 16 of the 32 bits you want to read out. Therefore, you will have to mask the output of the block RAM to select the appropriate portion of the 32-bits you read out. For example, if you want to execute a lb on an address ending in 01, you will want bits [23:16] of the 32 bits that you read out of block RAM (thus storing {24'b0, output[23:16]} to a register).

To take care of sb and sh, note that the `we` input to the imem and dmem modules is 4 bits wide. These 4 bits are a byte mask telling the block RAMs which of the 4 bytes to actually write to. If `we`={4'b1111}, then all 32 bits passed into the module would be written. However, if `we`={4'b1000}, then only bits [31:24] of the data would be written to the address: {addr_in, 2'b00}.

B Software Toolchain

A GCC MIPS toolchain has been built and installed in the cs150 home directory; these binaries will run on any of the p380 machines in the 125 Cory lab. The most relevant pieces of the toolchain are given below:

- `mips-gcc`: gcc for MIPS, compiles C code to MIPS binaries.
- `mips-as`: MIPS assembler, assembles code to MIPS binaries.
- `mips-objdump`: Displays contents of MIPS binaries in readable format.

The easiest way to use this toolchain will be to copy and rename the `example` project in the `software` directory of the skeleton files. After renaming the files for your new program, change the `TARGET` variable in the makefile.

There are several files in the example project, each with a specific purpose:

- `start.s`: This is an assembly file that contains the start of the program. It initializes the stack pointer then jumps to the `main` label. Edit this file to move the top of the stack.
- `example.ld`: This linker script sets the base address of the program. This needs to be set to the base address the program will be loaded into.
- `example.elf`: Binary produced after running `make`. Use `mips-objdump -D example.elf` to view the contents. You may want to redirect the output to a file or pipe it through less to examine the assembly.
- `example.mif`: Produced by the toolchain. Use this to initialize the block RAMs in ModelSim.
- `example.coe`: Produced by the toolchain. Use this to initialize the block RAMs during generation (coregen).

C Assembly Test

This section describes the contents of `software/src/asmtest`. You can test individual instructions with a program similar to the following example:

```
_start:

addiu $s7, $0, 0x0

# Test 1
li   $s0,      0x00000020
addiu $t0, $0, 0x20
addiu $s7, $s7, 1 # register to hold the test number (in case of failure)
bne  $t0, $s0, Error

j Done
Error:
# Perhaps write the test number over serial
Done:
# Write success over serial
```

In the given example, you have a few reference registers ($s0 in the example), which hold known values. You then use the instruction you want to test (or possibly in conjunction with instructions you've already tested earlier in the program) to create a copy of the reference value. You then check if the temporary register is equal to the reference register, and branch if it isn't.

Follow the directions from Appendix A and Appendix B to assemble your test program and use it in simulation. You will also need to write a Verilog testbench that instantiates the CPU and perhaps has helpful `$display` statements to help you debug your CPU.

C.1 Register File

Your register file should have two asynchronous-read ports and one synchronous-write port (positive edge). To test your register file, you should verify the following:

- Register 0 is not writable, i.e. reading from register 0 always returns 0
- Other registers are updated on the same cycle that a write occurs (i.e. the value read on the cycle following the positive edge of the write should be the new value).
- The write enable signal to the register file controls whether a write occurs (`we` is active high, meaning you only write when `we` is high)
- Reads should be asynchronous (the value at the output one simulation time (#1) after feeding in an input address should be the value stored in that register)

After you build your design, look for warnings in the report (`make report`) about the register file. Occasionally, the tools infer a block RAM rather than distributed (slice) RAM. This will not show up in simulation, but it will cause synchronous reads on hardware. To fix this, you can add a flag to your register file:

```
(* ram_style = "distributed" *) reg myReg...
```

D BIOS and Programming your CPU

We have provided a "BIOS" program in `software/bios150v3` that allows you to interact with your CPU and bootstrap into other programs over the serial interface. To use this, compile the program, rebuild your instruction and data memories with the generated `.coe` file, build your CPU, and impact it to the board. Then, as in lab 5, use screen to access the serial port:

```
screen $SERIALTTY 115200
```

Please remember to shut down screen using Ctrl-a shift-k, or other students won't be able to use the serial port!

If all goes well, you should see a '>' prompt after pressing return. The following commands are available:

- `jal <address>`: Jump to address (hex).
- `sw, sb, sh <data> <address>`: Store data (hex) to address (hex).
- `lw, lb, lhu <address>`: Prints the data at the address (hex).

In addition to the command interface, the bios also allows you to load programs to the CPU. Close screen using ctrl-a shift-k, and execute in the terminal:

```
coe_to_serial <coe_file> <address>
```

This stores the `.coe` file at the specified hexadecimal address. In order to write into both the data and instruction memories, remember to set the top nibble to 0x3. You also need to ensure that the stack and base address are set properly (See Appendix B).

E Git

If you have not yet configured your repository, please follow the instructions in Lab 5.

You should check frequently for updates to the skeleton files. To pull them into your repository, assuming you have correctly followed the configuration instructions, issue this command from a directory in your repository:

```
git pull staff master
```

F Protips

In previous iterations of this project, students have struggled with the following issues:

- **Off by one errors**. These occur in many different forms, but are usually the result of not thinking carefully about the timing of your design. It is important to understand the difference between synchronous and asynchronous elements. The synchronous elements in your design include the UART, Block RAMs for data and instruction memory, registers, as well as the register file (write-only!).

- **Memory mapped I/O**. As the name implies, you should treat I/O such as the UART just as you would treat the data memory. This means that you should assert the equivalent write enable (i.e. valid) and data signals at the end of the execute stage, and read in data in the memory stage. The CPU itself should not check the valid and ready signals; this check is handled in software.
- **Byte/halfword/word and endianness**. Read Appendix A.2 carefully, and ask questions if you are confused at all.
- **Incorrect control signals**. A comprehensive assembly test program will help you systematically squash bugs caused by incorrect control signals.