# Programming Assignment (Simulating Pipelined Machine)

Due 1: July 31 (Basic implementation)
Due 2: August 13 (Complete implementation)

In this assignment, you will become familiar with how a basic MIPS 5-stage pipeline works. You will be given a simulator that models an **unpipelined** processor that implements a small MIPS-like instruction set. Your assignment is to create a cycle-accurate simulator of a pipelined version of this processor. Your simulator will perform (1) data forwarding, (2) a simple branch prediction scheme, and (3) the pipeline interlocks to stall the pipeline when necessary. You will need a Linux environment to complete this assignment. You may use your own installed version on your computer if you use a Mac or a Linux OS, or you can work on the `grace` cluster. The course space is already created in the cluster and you can login with your directory ID and password (ssh yourDirectoryID@grace.umd.edu).

If you use a Windows machine and still want to work locally on your computer, you may work on a virtual machine environment.

## 1 Files

The first thing you should do is to check the files downloaded from ELMS. There are 8 files:

- `asm.c`
- `Makefile`
- `mips-small-pipe.c`
- `mips-small-pipe.h`
- `mips-small.c`
- `mult.s`
- `simple.output`
- `simple.s`

`asm.c` is an assembler for the reduced MIPS ISA which your simulator will implement (more about the assembler and the ISA later). `mips-small.c` is the C source file for a fully functional unpipelined simulator. `mips-small-pipe.c` is a C source code template that has some useful data structures and routines for a pipelined simulator. You can use this template to get started on the assignment. `mips-small-pipe.h` is a header file included by `mips-small-pipe.c`, and `Makefile` is a unix make file which will produce two binaries `asm`, `sim`, and `sim-pipe` from the source files, `asm.c`, `mips-small.c`, and `mips-small-pipe.c`, respectively.

In the remaining files, we have provided two example assembly programs. `simple.s` performs an arithmetic operation and `mult.s` multiplies two numbers. `simple.output` is the result of running `simple.s` through our pipelined simulator, and can be used to verify your simulator's output.

| Name | Format Type | Opcode | Func |
|---|---|---|---|
| LW | I-type | 0x23 | |
| SW | I-type | 0x2B | |
| BEQZ | I-type | 0x4 | |
| ADDI | I-type | 0x8 | |
| ADD | R-type | 0x0 | 0x20 |
| SUB | R-type | 0x0 | 0x22 |
| SLL | R-type | 0x0 | 0x4 |
| SRL | R-type | 0x0 | 0x6 |
| AND | R-type | 0x0 | 0x24 |
| OR | R-type | 0x0 | 0x25 |
| HALT | J-type | 0x3F | |

Table 1: Instruction encodings for a reduced MIPS ISA.

## 2 A Scaled-Down MIPS ISA

You will be simulating the MIPS ISA from Hennessy & Patterson, with some key differences. First, instead of a 64-bit architecture, you will implement a 32-bit architecture. In other words, all registers and data paths are 32 bits wide, and all instructions will operate on 32-bit operands – an integer variable should be big enough to hold an instruction on the grace cluster. Second, to keep your simulator simple, you will only be required to support a scaled-down version of the MIPS ISA consisting of 11 instructions. These instructions, along with their encoding, are given in Table 1. We will adhere to the MIPS instruction formats presented in Figure A.22 (Instruction layout for MIPS) of Hennessy & Patterson, with the exception that there is no "shamt" field in the R-type format, and instead the "func" field is 11-bits wide. Finally, although all immediate values in branch and jump instructions are already left-shifted by 2, so your BEQZ instruction **SHOULD NOT** perform the left shift in the implementation. A simple implementation of MIPS in pages C-31 through C-33 and Figure C.23 would give you some hints on pipeline implementation.)

Notice that all the instructions in Table 1 except for one exist in the normal MIPS ISA. These instructions behave exactly as described in the text (except they are 32-bit versions rather than 64-bit versions). The instruction we've added is the `HALT` instruction. As its name implies, when your simulator executes a HALT instruction, it should terminate the simulation. For more information about MIPS ISA, consult Section A.9 of Hennessy & Patterson.

## 3 asm: An Assembler for the Reduced MIPS ISA

We have provided an assembler, `asm.c`, so that you can assemble programs for your simulator. The `asm.c` file is fully functional, and you will not need to make any modifications to this file. Simply use the `Makefile` to make the binary `asm` from the `asm.c` source file – i.e., You would work on a Linux environment and type `make` to compile everything using the `Makefile`.

The format for assembly programs is very simple. A valid assembly program is an ASCII file in which each line of the file represents a single instruction, or a data constant. The format for a line of assembly code is:

```
label<tab>instruction<tab>field0<tab>field1<tab>field2<tab>comments
```

The leftmost field on a line is the label field which indicates a symbolic address. Valid labels contain a maximum of 6 characters and can consist of letters and numbers. The label is optional (the tab following the label field is not. After the optional label is a tab – i.e., – the tab character indicates if there is a label or not. Then follows the instruction field, where the instruction can be any of the assembly-language mnemonics listed in Table 1. After another tab comes a series of fields. All fields are given as **decimal** numbers. The number of fields depends on the instruction. The following describes the instructions and how they are specified in assembly code:

```
lw   rd      rs1     imm     Reg[rd] <- Mem[Reg[rs1] + imm]
sw   rd      rs1     imm     Reg[rd] -> Mem[Reg[rs1] + imm]
beqz rd      rs1     imm     if (Reg[rs1] == 0) PC <- PC+4+imm
addi rd      rs1     imm     Reg[rd] <- Reg[rs1] + imm
add  rd      rs1     rs2     Reg[rd] <- Reg[rs1] + Reg[rs2]
sub  rd      rs1     rs2     Reg[rd] <- Reg[rs1] - Reg[rs2]
sll  rd      rs1     rs2     Reg[rd] <- Reg[rs1] << Reg[rs2]
srl  rd      rs1     rs2     Reg[rd] <- Reg[rs1] >> Reg[rs2]
and  rd      rs1     rs2     Reg[rd] <- Reg[rs1] & Reg[rs2]
or   rd      rs1     rs2     Reg[rd] <- Reg[rs1] | Reg[rs2]
halt                         stop simulation
```

Note that in the case of the `beqz` instruction, PC-relative addressing is used (and again, your simulator should not perform the left-shift when computing the PC-relative branch target). For the `lw`, `sw`, and `beqz` instructions, the `imm` field can either be a decimal value, or a label can be used. In the case of a label, the assembler performs a different action depending on whether the instruction is a `lw` / `sw` instruction, or a `beqz` instruction. For `lw` and `sw` instructions, the assembler inserts the absolute address corresponding to the label. For `beqz` instructions, the assembler computes a PC-relative offset with respect to the label. After the last field is another tab, then any comments. The comments end at the end of the line.

In addition to instructions, lines of assembly code can also include directives for the assembler. The only directive we will use is `.fill`. The `.fill` directive tells the assembler to put a number into the place where the instruction would normally be stored. The `.fill directive` uses one field, which can be either a numeric value or a symbolic address. For example, "`.fill 32`" puts the value 32 where the instruction would normally be stored. In the following example, "`.fill start`" will store the value 8, because the label "start" refers to address 8 (remember that the MIPS architecture uses byte addresses).

```
        addi    1       0       5       load reg1 with 5
        addi    2       0       -1      load reg2 with -1
start   add     1       1       2       decrement reg1
        lw      3       0       var1    loads reg3 with value stored in var1
        addi    3       3       -1      decrement reg3
        sw      3       0       var1    put reg3 back (thus var1 is decremented)
        beqz    0       1       done    goto done when reg1==0
        beqz    0       0       start   back to start
        add     0       0       0
done    halt
        .fill   start                   will contain start address (8)
var1    .fill   32                      Declare a variable, initialized to 32
```

Try taking the above example and running the assembler on it. Enter the above assembly code into a file called "ex.s". Then type "asm ex.s ex.out". The assembler will generate a file "ex.out" which should contain:

```
(address 0x0):   20010005
(address 0x4):   2002ffff
(address 0x8):   00220820
(address 0xc):   8c03002c
(address 0x10):  2063ffff
(address 0x14):  ac03002c
(address 0x18):  10200008
(address 0x1c):  1000ffe8
(address 0x20):  00000020
(address 0x24):  fc000000
(address 0x28):  00000008
(address 0x2c):  00000020
```

The assembler assumes that all programs will be loaded into memory beginning at address 0x0.

# 4 Simulator without Pipelining

As described above, we have provided you with a simulator that simulates the same ISA, but which is not cycle accurate because it does not account for pipelining. The source code for this simulator is in `mips-small.c`. You can use this simulator to run assembly language programs prior to getting the pipelined simulator working.

Build the unpipelined simulator by typing "`make sim`" in the directory where you've copied the files we have provided. This will produce an executable `sim` which you can run. Try running this simulator on the program we have provided, `mult.s`. This program multiplies two numbers, specified in memory at the labels `mcand` and `mplier`, and stores the result at the label `answer`. Be sure to assemble the program before running the simulator. The sequence of commands you should give are:

```
make sim
asm mult.s mult.out
sim mult.out
```

When you run the simulator, a ton of messages will spew onto your terminal. The simulator we have provided dumps the state of the machine (including the contents of registers and memory) at every simulated cycle. Note that `mips-small.c` implements an unpipelined, single-cycle machine. So, each instruction completes in a cycle, which is *long* enough to process everything needed from fetching an instruction through updating the register file.

You can examine this output to see what the simulator is doing. To put these messages in a file so that you can actually read them, redirect the output of the simulator to a file. – e.g., on a C shell, by saying "`sim mult.out >! sim.output`".[1] Your pipelined simulator will spew similar messages, except it will also include the state of pipeline registers.

In addition to using the unpipelined simulator to run assembly programs, you should also look at the code in mips-small.c. The code is very simple, but you should make sure you understand it before moving on to building the pipelined simulator.

# 5 Assignment

Your assignment is to build a cycle-accurate simulator that accounts for pipelining. We have provided some code to get you started in `mips-small-pipe.c` and `mips-small-pipe.h`. This code serves two purposes: it will make your life a bit easier since you won't have to write the whole simulator from scratch, and it is meant to enforce some coding disciplines that ensure you actually simulate the details of the processor pipeline.

The following sections describe the assignment in more detail.

## 5.1 Simulate Pipelining

You will be simulating the basic MIPS pipeline depicted in Figure C.22 of Hennessy & Patterson. We have provided pipeline register data structures specified in the file, `mips-small-pipe.h`. **To enforce an implementation of your simulator that simulates pipelining, you are required to use these structures in their unmodified form.** In other words, each cycle simulated by your simulator must produce the correct values for all the fields in the pipeline register data structures.

[1] You may search Google with the keywords `input output redirection` if you have not used the input/output redirection on a Linux environment.

```
typedef struct IFIDStruct {
  int instr;
  int pcPlus1;
} IFID_t;

typedef struct IDEXStruct {
  int instr;
  int pcPlus1;
  int readRegA;
  int readRegB;
  int offset;
} IDEX_t;

typedef struct EXMEMStruct {
  int instr;
  int aluResult;
  int readRegB;
} EXMEM_t;

typedef struct MEMWBStruct {
  int instr;
  int writeData;
} MEMWB_t;

typedef struct WBENDStruct {
  int instr;
  int writeData;
} WBEND_t;

typedef struct stateStruct {
  int pc;
  int instrMem[MAXMEMORY];
  int dataMem[MAXMEMORY];
  int reg[NUMREGS];
  int numMemory;
  IFID_t IFID;
  IDEX_t IDEX;
  EXMEM_t EXMEM;
  MEMWB_t MEMWB;
  WBEND_t WBEND;
  int cycles; /* number of cycles run so far */
} state_t, *Pstate;
```

There are three small modifications in the above pipeline register data structures as compared with Figure C.22 in Hennessy & Patterson. First, we don't latch the "branch taken" signal in the EXMEM register. Instead, when your simulator computes the direction of the branch in the execute (EX) stage, you can directly modify the PC register from the execute (EX) stage. Second, the MUX in the WB stage has been moved to the MEM stage. Therefore, the MEMWB pipeline registers only latch two values, `instr` and `writeData`, instead of three values as indicated by Figure C.22 in the textbook. Finally, we have added an extra pipeline register after the WB stage. Instead of communicating internally through the register file, your simulator will always forward from the pipeline registers. For instance, in Figure C.7 of the textbook, there is a communication path from the DADD instruction to the OR instruction internally through the register file (this is achieved by writing the register file on the first half of the clock cycle, and reading the register file on the

second half of the clock cycle). In your simulator, this register communication will be replaced by forwarding between the WBEND pipeline registers to the execute stage of the OR instruction in cycle CC6.

As mentioned above, your simulator must produce all the values for the pipeline register data structures on a cycle-by-cycle basis. **In addition, your simulator must also use the following simulator loop code** (which is in `mips-small-pipe.c`):

```
while (1) {

  printState(state);

  /* copy everything so all we have to do is make changes.
     (this is primarily for the memory and reg arrays) */
  memcpy(&new, state, sizeof(state_t));

  new.cycles++;

  /* --------------------- IF stage --------------------- */

  /* --------------------- ID stage --------------------- */

  /* --------------------- EX stage --------------------- */

  /* --------------------- MEM stage --------------------- */

  /* --------------------- WB stage --------------------- */

  /* transfer new state into current state */
  memcpy(state, &new, sizeof(state_t));
}
```

You will insert the necessary code between the comments that perform the functions at each pipeline stage, but do not need to change any other part of this loop. Especially, **be careful not to change the output format of printState**.[2] Within each pipeline stage, the code will compute the new values of the pipeline register data structures (stored in the "`new`" structure) as a function of the old values (stored in the "`state`" structure). Then, after the code in all the stages have computed their values, the "memcpy" at the end of the simulator loop copies the values from the `new` structure into the `state` structure. This corresponds to the rising edge of the clock latching values into the pipeline registers for the next clock cycle.

## 5.2 Dealing with Data Hazards

Your simulator should deal with data hazards by **bypassing (or forwarding)** values from the pipeline register data structures. You can implement such data forwarding by checking for data dependences in the execute stage. For every instruction that enters the execute stage, you will have to check against the results produced by the previous 3 instructions (whose results can be found in the EXMEM, MEMWB, and WBEND pipeline registers). If there are no data dependences, then the execute stage should take its operands from the `readRegA` and `readRegB` IDEX pipeline registers. If there are data dependences, then the execute stage should instead copy the results from the pipeline register of the dependent instruction.

Most data hazards can be resolved using bypassing. However, there is one case that requires stalling the pipeline. As seen in Figure C.9 in the textbook, if an instruction uses a source register that is the destination register of an immediately preceeding `load` instruction, then the pipeline must be stalled for a single cycle. This can be implemented by checking for such a dependence in

[2] You are allowed to write helper functions in the C file if needed.

the decode stage and inserting a bubble (NOP instruction - see Section 5.4.2) instead of allowing the instruction to proceed. One special case is when a load from a location is immediately followed by a store (Figure C.8 in the textbook). The depending store can be bypassed by forwarding from the MEM/WB latch (loaded value) to the data memory write port of the memory stage (store instruction). However, for your implementation, you will not need to implement this; simply stall the pipeline one clock cycle for ALL load instructions that are followed by a depending instruction, even if the depending instruction is a store.

## 5.3 Dealing with Control Hazards

The pipeline you will be simulating will have a branch delay of 2 cycles since the branch direction is not resolved until the end of the execute stage. Instead of stalling the pipeline for 2 cycles after every fetched branch, your simulator should implement a "hybrid" static branch prediction scheme. In this scheme, you should predict taken for backward branches, and predict not-taken for forward branches. To implement this hybrid scheme, **test the immediate value in the fetch stage of every fetched branch instruction.** If the immediate value is positive, then fetch the next consecutive instruction on the following cycle. If the immediate value is negative, then fetch the branch target (computed as a PC-relative address using the offset) on the following cycle.

Make sure that your simulator validates the branch direction computed in the execute stage against the predicted direction. If the prediction was correct, no action should be taken. If the prediction was incorrect, then the simulator must squash the two instructions in the pipeline, which at this point will be in the IFID and IDEX pipeline registers. This squashing can be accomplished by replacing the incorrectly fetched instructions with NOP instructions and clearing all the other state in the IFID and IDEX pipeline registers.

## 5.4 Miscellaneous

### 5.4.1 Memory

As indicated by the `state_t` structure described in Section 5.1, there are two arrays that implement memory, an instruction memory array `instrMem` and a data memory array `dataMem`. This simulates the separate instruction and memory structures in the pipeline in Figure C.22 in the textbook. The code we have provided initializes both of these arrays to the same values read in from the assembly code file. In your simulator, you should read instructions from the `instrMem` array in the fetch stage, and write (read) values to (from) the `dataMem` array in the memory stage.

In the simulator code we have provided for you, the amount of memory simulated is determined by the size of the assembly program file that you read into the simulator. The `numMemory` field of the `state_t` structure is set to this size during initialization and is fixed for the duration of the simulation. This means that you must allocate all the memory your program will need statically in the assembly code file. The maximum size of the assembly program allowed is set by the `MAXMEMORY` constant declared in `mips-small-pipe.h` and is currently 16K words (64K bytes).

### 5.4.2 NOP Instruction

The MIPS ISA does not include an explicit NOP (null operation) instruction, and neither does our reduced MIPS ISA. However, the NOP instruction is needed to initialize all the instruction fields of the pipeline register data structures, and to replace or "squash" those instructions in the pipeline following a branch misprediction. In your simulator, you will use the "`add 0 0 0`" instruction as a NOP instruction (since this instruction does not effect any state). We have already declared this as `NOPINSTRUCTION` and used it in the initialization code.

### 5.4.3 Halting

When your simulator encounters the `HALT` instruction, it should end the simulation. Notice, however, that it is incorrect to halt as soon as the HALT instruction is fetched since this code may be

speculative (dependent on a yet to be verified branch). If the branch was mispredicted, then you definitely do not want to halt. Another issue is that any `SW` instructions which are farther along the pipeline than the `HALT` instruction must be allowed to complete before the machine is stopped. To solve these problems, halt the machine when the `HALT` instruction reaches the MEMWB pipeline register. This ensures that all previously executed instructions have completed.

### 5.5 Grading

Your work will be graded based on the correctness of the required functionality. In other words, for any valid reduced MIPS ISA program, your simulator should accurately simulate the pipeline and produce the correct values for all the pipeline registers at each cycle. We will be using our own programs to validate your simulator, so make sure you test your simulator on several test cases (in other words, you should write a few extra assembly programs to test your simulator). To assist in grading, use the simulator loop template and the `printState` routine that we have given you in their unmodified form. Each time you call `printState`, it should print in the format that we have provided, and you should only call `printState` once every iteration of the simulator loop (you are free to use `printState` as much as you want when debugging, but take all the extra `printState` calls out of your code before handing in your assignment). Finally, compare the output of your simulator on the `simple.s` program to output given in `simple.output`. There should be no differences (i.e., running `diff simple.output <your output file>` should yield no differences).

### 5.6 Academic Honesty

Do not allow any other student to see any of your code. You may however discuss the assignment in general terms, with other students, only for the clarification purpose. The line between clarification purpose discussion and academic dishonesty. You should never show your code to others or reuse code you see on any on-/off-line sources. Also, do not discuss too much implementation details during discussions. Both of you could write very similar code in the end. If copying or excessive collaboration is detected in your submissions, the matter will be referred to the Office of Student Conduct.

## 6 Submission

You will submit `mips-small-pipe.h` and `mips-small-pipe.c` to the submit server. Public tests will be distributed soon.