This folder contains three simulated implementations of a subset of the MIPS ISA. These simulations are not intended to show how the MIPS ISA actually is implemented - rather they are intended to familiarize students with various implementation strategies using an ISA they are already familiar with. (I use the MIPS ISA for teaching assembly language.) Two of the strategies are similar - but not identical - to actual strategies that have been used in commercial implementations. One is definitely not the way the MIPS ISA is actually implemented.

Subset of the MIPS ISA

These simulations implement a small subset of the MIPS ISA, including only the following instructions:

R-Type instructions:	ADD, ADDU, AND, NOR, OR, SLL, SLLV, SLT, SLTU, SRA, SRAV, SRL, SRLV, SUB, SUBU, XOR (However, overflow is not detected by the "Non-U" versions of arithmetic instructions, so both forms are equivalent - e.g. in the simulation both ADD and ADDU behave identically)
Immediate instructions:	ADDI, ANDI, LUI, ORI, SLTI, XORI
Load-Store instructions:	LW, SW
Branch/Jump instructions:	BEQ, BNE, J, JAL, JR
MIPS IV ISA:	MOVN, MOVZ

Implementation Strategies

- Multicycle: This implementation is included only for pedagogical purposes. Actual MIPS implementations are pipelined, and exhibit behaviors (e.g. delayed branch and delayed load) which this implementation does not. However, understanding this implementation can help the student to understand key concepts before trying to understand a pipelined implementation.

  Most instructions are executed in four clock cycles (though some use only the first two). The file MulticycleMIPSMicroArchitecture.pdf depicts the microarchitecture used for this implementation. The file file MulticycleMIPSRTL.pdf shows the microoperations performed on each clock cycle for each instruction (except MOVN and MOVZ - see the Lab discussion below). Microoperations are controlled by a 17-bit control word, with twelve groups of 1, 2, or 3 bits controlling the various data paths and enabling the loading of various registers.

  This simulation includes three different control units, with one being selected at any given time. (The default when it first starts is Manual).

  - Manual: the control word is determined by manually setting the values of the various fields. Using this, it is possible to see how an individual instruction is executed by manually setting the control word to the value specified by the RTL for each cycle of the instruction. (Of course, this is only useful for pedagogical purposes unless one were interested in a CPU running at human speeds :-)).
  - Hardwired: a counter keeps track of the cycle number (0, 1, 2, or 3). Each bit of the control word is a function of the current cycle number , the contents of the Instruction Register, and (in some cases) a comparator that compares two registers for equality. On a real machine, these would be calculated by a network of gates, but in the simulation each bit is calculated by boolean functions realized in software.
  - Microprogrammed: the bits of the control word are part of a microprogram stored in the control unit. (Of course, RISC architecture is kept simple to avoid the need for microprogramming, but this alternative shows how microprogramming could be used.)

- Pipelined (without interlocking): This uses essentially the same microarchitecture as for the multicycle simulation, but broken into four stages, and requires four clocks to complete most instructions. (A three clock implementation would be possible by using both edges of the clock, but for pedagogical reasons the simplicity of this approach seems preferable.) At any given time, four successive instructions are being executed, one in each stage of the pipeline. There are three Instruction Registers, one each for Stages 1, 2, and 3.
  - Stage 0 performs instruction fetch. Stage 0 reads (but does not write) the Program Counter. It uses a dedicated read-only memory port. It is not governed by any Instruction Register, but writes the Instruction Register for Stage 1.
  - Stage 1 performs two tasks: updating the program counter (either by incrementing the current value or by loading a branch/jump address), and loading values into the input registers of the ALU. It therefore reads and writes the Program Counter, reads (but does not write) registers in the Register Set, and writes (but does not read) the ALU input registers. At the end of the cycle, it copies its Instruction Register into the Instruction Register for Stage 2.
  - Stage 2 performs a computation in the ALU. It reads (but does not write) the ALU input registers, and writes (but does not read) the ALU output register. At the end of the cycle, it copies its Instruction Register into the Instruction Register for Stage 3.
  - Stage 3 performs at most one of three tasks, depending on the instruction it is executing: storing the value computed by the ALU into a register, reading a value from memory into a register, or writing a value from a register into memory. (The latter two operations use the output of the ALU to specify the address). It therefore reads the ALU output register, and may either read or write a register in the Register Set. It also has its own read-write memory port.

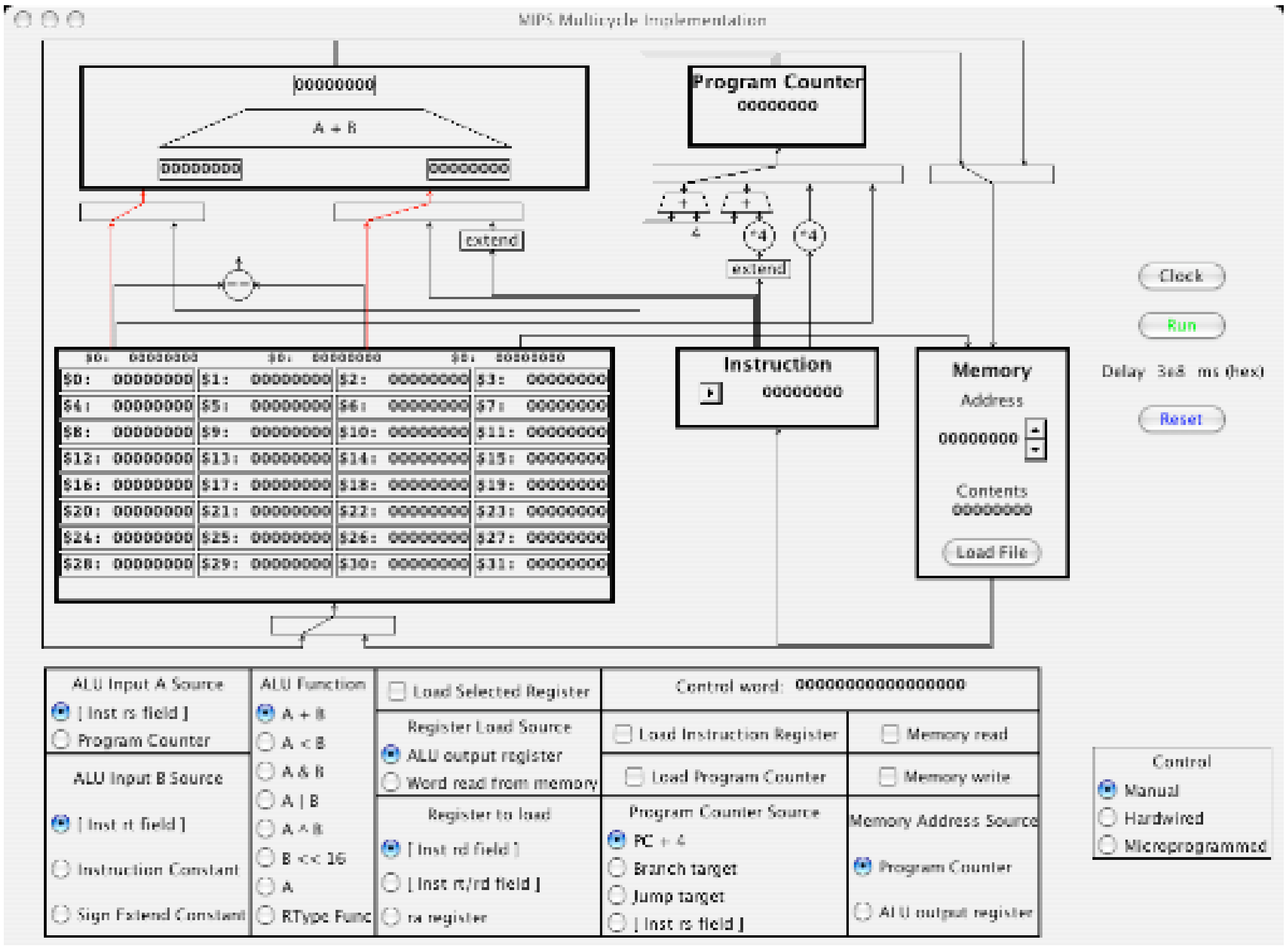
  This simulation illustrates the two sorts of inter-stage dependency (hazard) that can occur in an ISA like MIPS: branch dependencies and data dependencies.
  - It exhibits one cycle delayed branch.
  - It utilizes forwarding to resolve the data dependency between a computational instruction that stores a result into a register and an immediately following instruction that uses the value in that same register. That is, when appropriate, Stage 1 loads an ALU input register with the value being computed by the ALU in Stage 2, rather than using the (not-yet-updated) value of a register.
  - It also utilizes forwarding to reduce to just one cycle (but not to eliminate) the data dependency between a load instruction that reads a value from memory into a register and a following instruction that uses the value in that same register. That is, when appropriate, Stage 1 loads an ALU register with the value being read from memory by Stage 3 (by the instruction two ahead of it), rather than using the (not-yet-updated) value of a register.
  - However, the instruction <u>immediately</u> after a load instruction will get the old value of the register being loaded if it tries to use it; only an instruction two after a load instruction can use the value loaded. (This amounts to one cycle delayed load.)

- Interlocked: this implementation is essentially the same as the pipelined implementation, except that it uses interlocking to eliminate data hazards (and thus does not require delayed load though it still requires delayed branch). This is the strategy used for MIPS implementations since the MIPS III ISA. Moreover, for pedagogical purposes, understanding how interlocking can be used can help the student to understand superscalar systems where some form of interlocking is generally required.

Using the Simulations

The three simulations exist in three executable jar files: mmips.jar (Multicycle), pmips.jar (Pipelined without interlocking), and imips.jar (Interlocked).

The multicycle simulation presents the user with a graphical user interface that looks like this:



$0:	00000000	$1:	00000000	$2:	00000000	$3:	00000000
$4:	00000000	$5:	00000000	$6:	00000000	$7:	00000000
$8:	00000000	$9:	00000000	$10:	00000000	$11:	00000000
$12:	00000000	$13:	00000000	$14:	00000000	$15:	00000000
$16:	00000000	$17:	00000000	$18:	00000000	$19:	00000000
$20:	00000000	$21:	00000000	$22:	00000000	$23:	00000000
$24:	00000000	$25:	00000000	$26:	00000000	$27:	00000000
$28:	00000000	$29:	00000000	$30:	00000000	$31:	00000000

ALU Input A Source: [✓ [Inst rs field]] Program Counter

ALU Input B Source: [✓ [Inst rt field]] Instruction Constant Sign Extend Constant

ALU Function: [✓ A + B] A < B A & B A | B A ^ B B << 16 A RType Func

☐ Load Selected Register

Register Load Source: [✓ ALU output register] Word read from memory

Register to load: [✓ [Inst rd field]] [Inst rt/rd field] ra register

Control word: 000000000000000000

☐ Load Instruction Register

☐ Memory read

☐ Load Program Counter

☐ Memory write

Program Counter Source: [✓ PC + 4] Branch target Jump target [Inst rs field]

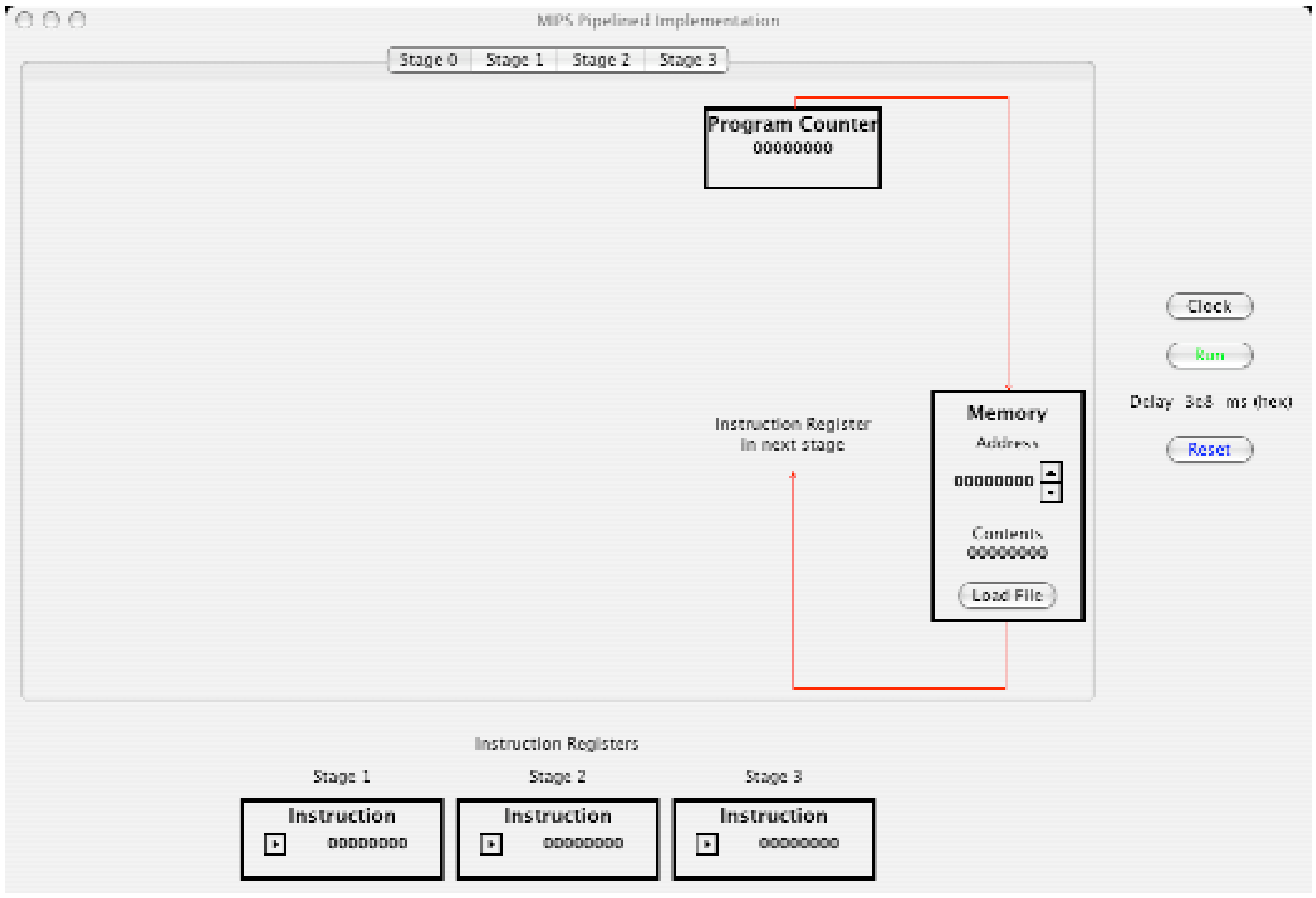
Memory Address Source: [✓ Program Counter] ALU output register

Control: [✓ Manual] Hardwired Microprogrammed

- The upper-left portion of the screen displays the microarchitecture. Active data paths are shown in red; others are black. For each MUX, a path is shown from the selected input to the output. The display above shows manual control selected. Changing the selection in one of the groups of radio buttons, or checking or unchecking a checkbox, will cause appropriate bits in the control word to change. This will, in turn, be represented in the data paths.
- The value in any register can be edited by double-clicking it. The Instruction Register can also be expanded to show the various fields (as determined by the op-code) by clicking its disclosure triangle. When individual fields are shown, they can also be edited by double-clicking. The contents of various memory locations can be shown by either editing the memory address or clicking the up and down arrow buttons. The contents of the memory location at the selected address can be edited by double-clicking it. It is also possible to load a text file into memory - see below for discussion of the format of this file. (Equivalently, memory can also be loaded by choosing the Load option in the program's File menu).

- A group of radio buttons at the lower right allows selection of either Hardwired or Microprogrammed control in place of Manual control. In these cases, the control word is automatically calculated on each cycle. using the contents of various registers. The state of the selected control unit is displayed on the lower left of the screen.
- A group of buttons in the upper-right allows one to simulate execution. Clicking the Clock button will issue a single clock, which will result in the microoperation specified by the control word being carried out. If Hardwired or Microprogrammed control is selected, a new control word will also be calculated using the new state of the system. Clicking Run will cause clocks to be issued automatically. (This will not normally produce meaningful results if Manual Control is selected, though). The delay between clocks can be set by double-clicking the delay field, which will pop up an editor for it. While the simulation is running, the Run button will be changed to a Stop button, which will terminate automatic issuing of clocks. Clicking Reset will restore all registers to their initial state (0), but will not alter the contents of memory.

The pipelined and interlocked implementations present the user an interface like this:



Tabs at the top allow the user to select separate panes for each Stage. Each Stage exhibits only the components of the microarchitecture used by that stage, though all three instruction registers are always shown at the bottom of the screen. Editing of register contents and memory, disclosure of Instruction Register contents, and simulated execution work the same way as with the Multicycle implementation (though instruction register disclosure is a bit unaesthetic!). A form of hardwired control is always used for each stage.

These versions use blue text to report situations where forwarding or interlocking is being done.

Format of Program Files

It is possible to load a MIPS program into memory by using either the Load File button on the displayed Memory or the Load option in the File menu. Each non-comment line in the file will contain one of two things:

- A single 32-bit word to be loaded into memory (written in hexadecimal).
- An "@" character, followed by a hexadecimal number which represents the address where subsequent words are to be loaded.

Words contained in the file are loaded into successive memory location, but since each represents a 32 bit word, addresses increase by 4 on each step. If no starting address is specified, the first word goes into location 0, the second into 4 ... It is also possible (and desirable) to embed comments in the file, beginning with "#". (If the first nonblank character on a line is "#", the entire line is ignored.)

The following is an example of a simple program file. Note that the assembly language code given in the comments is strictly for the benefit of the human reader; the translation of the code into machine language had to be done by hand when the file was created.

```
# This program adds 1 to the contents of memory location 1000
8c021000    # lw $2, 1000($0)
20420001    # addi $2, $2, 1
ac021000    # sw $2, 1000($0)
1000ffff    # b .
@1000
2a                  # Starting value at 1000 is 2a
```

Building from Sources

Although the executable jar files run correctly as they stand, it may be necessary to rebuild the system from source. All three versions share a common source code base, though some files are unique to a specific implementation. The makefile in the root folder can be used to build the desired version(s) - see the comments in that file. Three test programs are provided in the `programs/` folder for "sanity-checking" the build - one for each implementation. Each will set registers $1 .. $15 to 1, 2 ... f, and then will go into an infinite loop. The appropriate test can be executed easily by loading it, setting the delay to 0 and pressing Run; the correct values should then appear in the first 16 registers almost immediately, with registers $16 .. $31 unchanged.

Laboratory Exercise

I have used this simulation as the basis for a laboratory exercise, in which students complete a partially-written version of the hardwired control unit for the multicycle implementation by writing much of the code for calculating the bits of the control word. Materials from this lab are in the `Lab/` folder. The lab has two parts. In the first, students are given a skeleton file which implements only the instruction fetch step on Cycle 0. They must complete code corresponding to the RTL in MulticycleMIPSRTL.pdf. In part two, they write RTL for two new instructions (taken from the MIPS IV ISA): MOVN and MOVZ. Thought these instructions are actually implemented by the complete simulation, they are not included in the RTL. In lab, students must create RTL for these instructions and then write code to implement it. The folder contains a copy of a lab assignment, plus other files needed for the lab. Note that it is also necessary to create a "crippled" version of the simulation (lacking the hardwired control code) as described in the README file.

Test files are provided to make testing student code easy. The first file (Test1) will exercise part I of the lab is described under Building from Sources above. The second file (Test2) will exercise MOVN and MOVZ. Upon successful execution, it will set registers $1 .. $3 to 2a2a2a2a and leave other registers unchanged. Of course, the Part II program should also be able to correctly execute TestProgramMulticycle as well.