

School of Mathematics, Statistics, and Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz



MPSIM

A MIPS Architecture Pipeline Simulator

Technical Reference
Version 1.2/May 2007

Authors:

Dr. Pavle Mogin
Mark Pritchard

Abstract

This document contains technical information about MIPS Pipeline Simulator. The simulator has been made to emulate five stage RISC (reduced instruction set computer) architecture in accordance with the book Computer Organization and Design – The Hardware/Software Interface by D.A. Patterson and J. L. Hennessy. The simulator is written in Java.

Table of Contents

1. Who Made the Simulator	3
2. Supported MIPS Instructions	3
3. Pseudo Instructions	3
4. User's Interface	4
4.1. Window Panes	
4.2. Fields	
4.3. Buttons	
5. Program Compilation	7
6. Hazards in the MIPS Pipeline Simulator	8
6.1. What Makes Hazards Occur?	
6.2. How to Avoid Hazards?	
7. Modes of Operation	8
7.1. All Mode Buttons Off	9
7.2. Bubbles Button On	10
7.3. Forwarding Button On	11
8. How to Use the Simulator?	12

1. Who Made the Simulator

The initial design of the MIPS Pipeline Simulator was made by Michael Chamberlain and John Elmore at Georgia Institute of Technology, USA in late nineteen nineties. Also, some unspecified alterations to the original version were made by Curt Hill from the Valley City University, North Dakota, USA in early 2000. To produce the current version, Pavle and Mark made a great number of major changes and improvements. These changes were mainly made towards making the simulator mimic the MIPS computer architecture as faithfully as possible and to the extent that is needed by the COMP 203 Computer Organization course. Adding a considerable number of new features and functionalities also incurred a corresponding number of additions and interventions in Java code.

2. Supported MIPS Instructions

The simulator supports a relatively large number of MIPS instructions. The most important instructions supported are:

R-Type:

ADD, SUB, AND, OR, SLL, SRL, SLT;

I-Type:

ADDI, SUBI, ANDI, ORI, SLLI, SRLI;

Branch:

BEQ, BNE;

Memory:

SW, LW.

It should be noted that the simulator is not **case sensitive**. So, add and ADD are both plausible op codes.

There is a restriction, not present in the MIPS assembly language. The simulator does not allow using **register zero** (\$0) as the destination register of arithmetic and logic instructions.

Branch **labels** are supported. Also, here is an introductory instruction for people who don't want to use labels, but want to calculate an exact word offset to the target address (program line where to branch).

Let ba be the address of a branch instruction, sn the stage number in which the PC is updated when the branch is taken, ta the target address, and os the word offset of the branch instruction. Then, in a general case,

$$ba + 4*sn + os = ta,$$

so the branch word offset os is

$$os = (ta - ba - 4*sn)/4.$$

Computation of branch offset also depends on the mode of operation. The specific instructions for calculation of branch offsets can be found in section 6.

Comments start with the standard hash (#) sign.

There is no practical restriction on the program **length**.

3. Pseudo Instructions

There are two special pseudo instructions supported. These are NOP and END. The NOP pseudo instruction replaces an instruction that does nothing, like a real MIPS instruction `or $0, $0, $0`. The NOP instruction should be used in programs that are executed on a `data_path`, which does not insert bubbles automatically.

The END pseudo instruction is a replacement of the MIPS `syscall` with the parameter value of 10. It denotes to the simulator that the program end has been reached. When the END pseudo instruction comes in the ID stage of the pipeline `data_path`, it causes a pipeline stall, prevents advancing the program counter, flushes the content of the IF stage (whatever it contained), and starts inserting bubbles to clear the pipeline. Instructions preceding the END instruction are normally executed.

There is also a system generated pseudo instruction `INVALID`. The `INVALID` instruction is entered into the IF stage when the PC addresses a blank word in the instruction memory (a non existent code line in the program). It is not intended for use in MIPS programs. Its only aim is to clearly designate a wrong program control design.

4. User Interface

The user interface of the MIPS Simulator is given in Figure 1. The GUI contains a number of windows, fields and buttons, whose short description follows.

4.1. Window Panes

Instruction Memory – Contains assembly programs. Assembly programs are displayed in lines mimicking memory cells. Each line contains a word address and an assembly language instruction. Comments are also made visible, as the contents of the memory.

There is no strict limit on the number of programs and program lines that can be simultaneously loaded in the Instruction Memory.

Register File – Contains 32 registers each 32 bits long. Each register starts with `Ri :`, where $i = 0, 1, \dots, 31$ is the register's address, and has a content, which is represented as a decimal integer.

Data Memory – Contains data. The current version of the simulator contains Data Memory with the size of only 32 words. Each word starts with `MEMi :`, where $i = 0, 4, 8, \dots, 124$ is a byte address, and has a content, which is represented as a decimal integer.

Messages – Contains explanatory messages produced by the simulator. These are mainly about actions undertaken by the `data_path` control as a result of hazard detection. Also, there appear messages regarding the use of the simulator.

Stages – There are five stage windows: Fetch, Decode, Execute, memory, and Write Back, which correspond to the five MIPS pipeline stages. Each stage shows the address and the assembly instruction it is working on, and the results of actions undertaken.

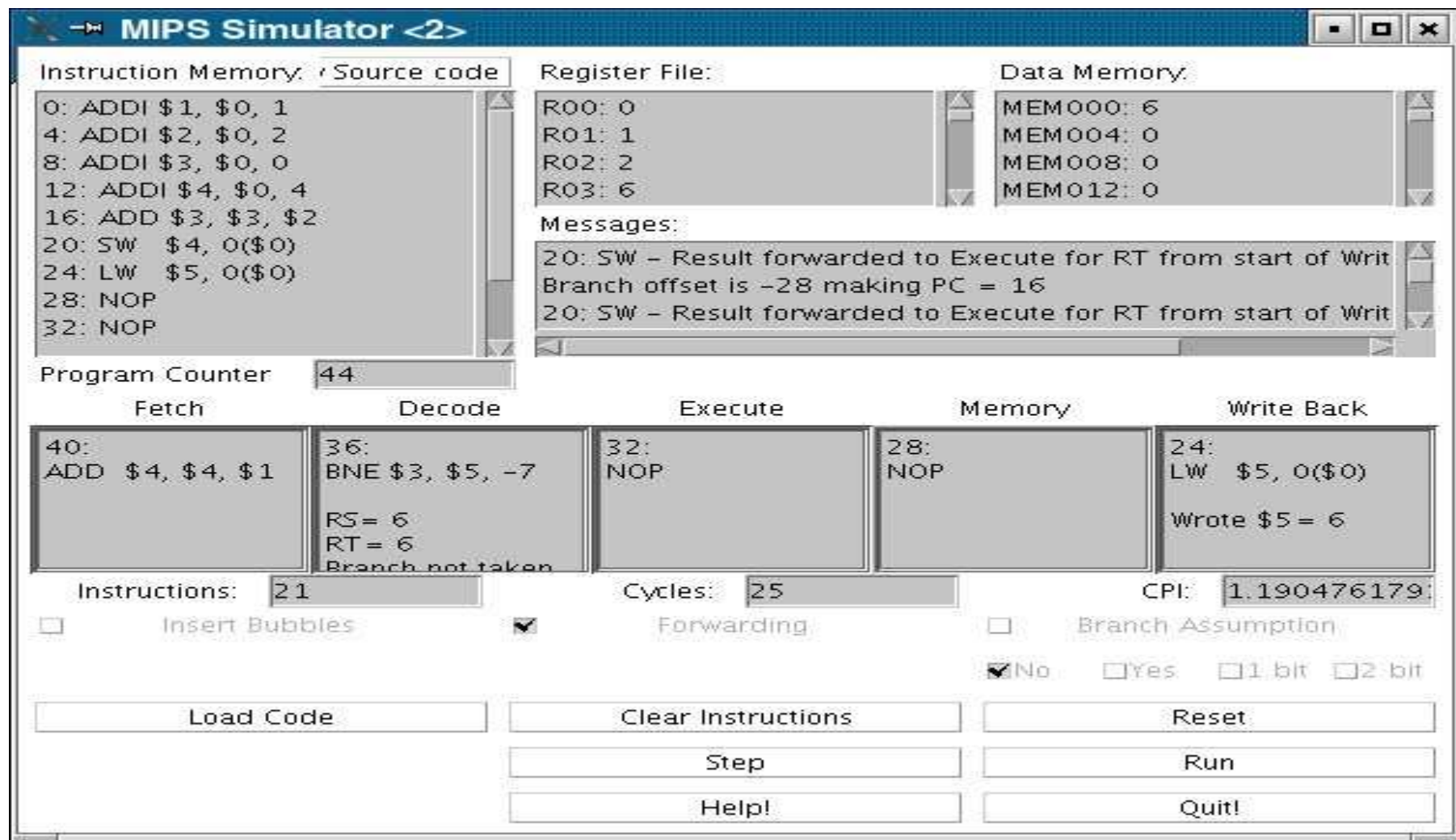
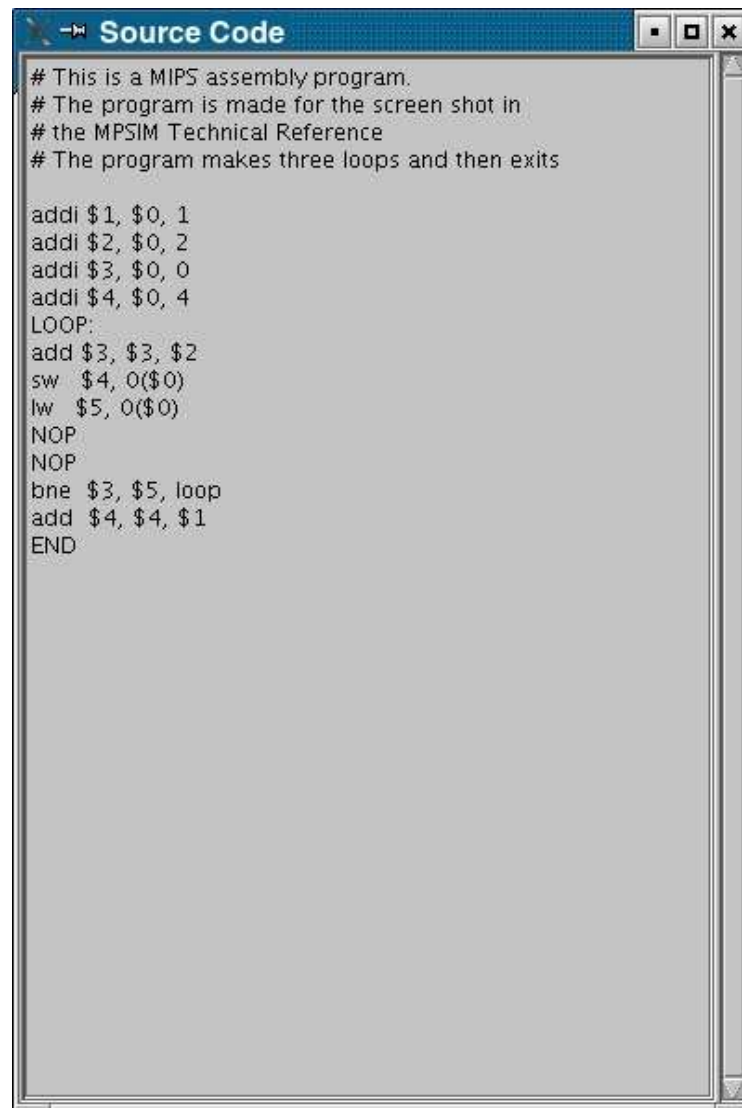


Figure 1.

Check Your Settings – This is a pop up pane that warns a user to check mode button settings when he/she presses Step or Run button. This warning is aimed to prevent execution of a program with a wrong setting of mode buttons, which would most probably lead to a program crash. The program will not start executing until the warning is acknowledged.

Source program pane – This pane pops up when the button View Source Code is pressed. The pane contains the source code version of the program in Instruction Memory. The source code version of the program in Figure 1 is shown in Figure 2.



```
# This is a MIPS assembly program.
# The program is made for the screen shot in
# the MIPSIM Technical Reference
# The program makes three loops and then exits

addi $1, $0, 1
addi $2, $0, 2
addi $3, $0, 0
addi $4, $0, 4
LOOP:
add $3, $3, $2
sw $4, 0($0)
lw $5, 0($0)
NOP
NOP
bne $3, $5, loop
add $4, $4, $1
END
```

Figure 2.

4.2.Fields

Program Counter – Contains the address of the instruction to be fetched next.

Instructions – A small field containing the number of instructions executed so far. Automatically inserted bubbles are not counted as instructions.

Cycles – Another small field containing the number of processor cycles used so far for execution of instruction, whose number is in the Instruction field. The number of cycles represents the most objective measure of the efficiency of the execution of a program. The smaller this number the better use of the pipeline while executing different versions of the same program.

IPS (Instructions per Cycle) – The third small field containing the ratio between the number of completed instructions and the number of cycles used for their execution. This is another measure of the quality of the pipeline utilisation. The smaller this value the better a program was able to take advantage of the pipeline parallelism while executing the same number of instruction as another one.

4.3. Buttons

View Source Code – Shows source code of the program in Instruction Memory.

Bubbles – Enters the simulator into a mode of operation where hardware bubbles are inserted between instructions that are at risk of a hazard.

Forwarding – Enters the simulator into a mode of operation where hardware prevents hazards by forwarding.

- **Branch Assumption** - Enters the simulator into a mode of operation where a branch not taken assumption has been made.

Load Code – Enables loading a program into Instruction Memory. If the desired program is not in the current directory, allows for a directory search.

Clear Instructions – Clears everything and leaves Instruction memory “empty”.

Reset – Clears Message window, Register File, and Data Memory, suspends any execution in progress, but does not clear instruction memory.

Step – Enables stepwise execution of a program.

Run – Executes the current program till the end.

Help – Displays help information.

Quit – Exits the simulator.

5. Program Compilation

The simulator compiles MIPS assembly programs. By compilation, the simulator:

- Removes empty lines and comments from the source program,
- Inserts memory addresses,
- Computes branch offsets, and
- Replaces branch labels by branch offsets.

Branch offsets are computed according to current setting of mode buttons, if that setting is changed as result of a warning made by Check Your Settings pop up pane, the branch offsets are recomputed automatically on the flight.

6. Hazards in the MIPS Pipeline Simulator

Because MIPS Pipeline Simulator simulates RISC architecture pipeline data_path it also simulates hazards. If one submits to MIPS Pipeline Simulator a program, which contains hazards, the program may crash or compute wrong answer. In fact, in most modes of operation it is programmer's responsibility to ensure that her/his program does not contain any hazards.

6.1. What Makes Hazards Occur?

There are two main situations in which hazards can occur. The first and most common situation is when one instruction updates some state in the processor (e.g. by writing value to a register). The hazard occurs because the second instruction tries to read the state before the first instruction has had time to finish writing back. The first instruction can be: an arithmetic, a logic, or a load word instruction, while the second instruction may be: an arithmetic, a logic, a store word, or a load word instruction.

The second situation is the consequence of the fact that a number of instructions directly following a taken branch instruction will be executed, although they should not.

6.2. How to Avoid Hazards?

Read: chapter 6 of the textbook, lecture notes on pipelining and hazards, and this manual.

7. Modes of Operation

The considered MIPS pipeline simulator simulates five different pipeline data_paths modes. These will be called:

- Basic,
- Bubbles,
- Bubbles with branch assumption,
- Forwarding, and
- Forwarding with branch assumption.

The simulated data_path mode is determined using mode buttons. There are three mode buttons:

- Bubbles,
- Forwarding, and
- Branch Assumption.

When all three mode buttons are off, the program simulates the basic pipeline data_path that relies completely on compiler (or student) control of hazards and has no performance enhancements in the case of hazards.

When the Bubbles button is on, the program simulates a pipeline data_path that does hazard detection, inserts bubbles, and stalls the pipeline.

When the Forwarding button is on, the program simulates a pipeline data_path that does hazard detection, does forwarding, makes a branching decision and possible Program Counter (PC) update within the second (ID) stage, and does not insert any bubbles and does not stall the pipeline.

When the Branch Assumption button is on, the program simulates a branching strategy where the **branch not taken** assumption is made. If a branch is actually taken, an appropriate number of instructions following the branch instruction are flushed.

The number of instructions flushed depends on whether Bubbles or Forwarding buttons is on.

When either Bubbles or Forwarding button is on, the program simulates a pipeline data_path where the Register File (RF) is *written* in the first half and *read* in the second half of a processor cycle. Otherwise, when both of these buttons are off, the program simulates a data_path where RF is *read* in the first half and *written* in the second half of a processor cycle.

If the Forwarding button is on, Bubbles button has to be off. The Branch Assumption button can be *on* only if either Bubbles or Forwarding button is on, but it can be off even if one of Bubbles or Forwarding buttons is on.

7.1.All Mode Buttons Off

When the all button (Bubbles, Forwarding, and Branch Prediction) are off, the program simulates a pipeline data_path that relies completely on compiler (student) control of hazards and has no performance enhancements in the case of hazards.

The table below describes the behaviour of the particular MIPS instructions in the case when all the buttons are off.

Instruction	Data Read	Data Written	Comment
R-type	From RF at the start of the stage 2 (ID)	Into register file at the end of the stage 5 (WB)	
Branch	From RF at the start of the stage 2 (ID)	If taken, the target address written into PC in the stage 4 (MEM)	The decision made in the stage 3 (EX) The target address computed in the stage 4 (MEM)
Lw	From RF at the start of the stage 2 (ID)	Into register file at the end of the stage 5 (WB)	Data read from the main memory in the stage 4 (MEM)
Sw	From RF at the start of the stage 2 (ID)	Into the main memory in the stage 4 (MEM)	

The number of nops needed between an R-type instruction (i_1) and a consecutive instruction (i_2) needing its result data is computed by subtracting the stage number when the instruction i_2 needs data from the stage number when the instruction i_1 writes data into the register File.

The number of nops needed after a branch instruction is computed by subtracting the IF stage number from the stage number when the branch instruction updates the PC.

Let ba be the address of a branch instruction ta be the target address, and os be branch word offset. Since sn the stage number in which the PC is updated is four, the branch word offset is computed in the following way:

$$os = (ta - 16 - ba) / 4.$$

7.2. Bubbles Button On

When the Bubbles button is on, the program simulates a pipeline data_path that does hazard detection, inserts bubbles, and stalls the pipeline. Also, the pipeline has a special register file where registers are written in the first half of a processor cycle, and read in the second half of a processor cycle. Hence, programmers are relieved of inserting nops to prevent hazards, and there is a hardware performance enhancement in the case of hazards.

The decision whether to take a branch or not is made in the third (EX) stage. The branch target address is calculated in the fourth (MEM) stage, and if the branch taken, the PC is also updated in the fourth (MEM) stage.

If the Branch Assumption button is off, then a branch instruction op code in the second (ID) stage induces a pipeline stall, the simulator stops the PC from advancing, to avoid reading instructions that perhaps should not be executed, and inserts three bubbles in the form of nops after the branch instruction (until it reaches the fourth (MEM) stage).

7.2.1. The Summary Table

The table below describes the behaviour of particular MIPS instructions in the case when the Bubbles button is on.

Instruction	Data Read	Data Written	Comment
R-type	From RF at the end of the stage 2 (ID)	Into RF at the start of the stage 5 (WB)	
Branch	From RF at the end of the stage 2 (ID)	If taken, the target address written into PC in the stage 4 (MEM)	The decision made in the stage 3 (EX) The target address computed in the stage 4 (MEM)
Lw	From RF at the end of the stage 2 (ID)	Into RF at the start of the stage 5 (WB)	Data read from the main memory in the stage 4 (MEM)
Sw	From RF at the end of the stage 2 (ID)	Into the main memory in the stage 4 (MEM)	

The number of bubbles in the form of nops that will be inserted automatically between an R-type instruction (i_1) and a consecutive instruction (i_2) needing its result data is two.

The number of bubbles in the form of nops that is inserted automatically after a branch instruction, if the Branch Assumption button is off, is three.

Let ba be the address of a branch instruction ta be the target address, and os be branch word offset. Since the branch instruction prevents the PC from advancing, the branch word offset is computed in the following way:

$$os = (ta - 4 - ba) / 4.$$

This is the only mode of operation of the pipeline data_path that does not require any programmer's intervention in program code to produce a correct result (except computing a correct branch offset).

7.2.2. Branch Assumption On

If the Branch Assumption button is on, it means that a **branch not taken** assumption is made. So, no bubbles are inserted after a branch instruction. The decision whether to take a branch or not is made at the end of third (EX) stage. Moreover, if a branch is taken, say in cycle c , then:

- PC will be augmented by 4 at the start of the cycle $c + 1$,
- IF, ID, and EX stages will be also flushed at the start of the cycle $c + 1$,
- PC will be updated to the target address at the end of the cycle $c + 1$, and
- The instruction at the target address will be fetched in the cycle $c + 2$.

The instructions flushed will be not executed and thus not counted.

If the branch is not taken, instructions following it will be executed.

Let ba be the address of a branch instruction ta be the target address, and os be branch word offset. Since sn the stage number in which the PC is updated is four, the branch word offset is computed in the following way:

$$os = (ta - 16 - ba) / 4.$$

7.3. Forwarding Button On

When the Forwarding button is on, the program simulates a pipeline data_path that does detection of those hazards that can be resolved by forwarding, performs sole forwarding, but does not insert any bubbles. Also, the pipeline has a special register file where registers are written in the first half of a processor cycle, and read in the second half of a processor cycle. Hence, programmers are relieved of inserting nops to prevent hazards that can be resolved by forwarding, and there is a hardware performance enhancement in the case of hazards.

The branch target address is calculated in the second (ID) stage. The decision whether to take a branch or not is also made in the second (ID) stage, and the PC is updated in the second (ID) stage, as well. The decision whether to branch or not is made using a special hardware unit (comparator) consisting of 32 exclusive OR gates and an AND gate. The data that are used to make a branch decision is needed at the start of the second (ID) stage. This can cause a problem, if an instruction just preceding a branch computes the data needed for comparison and decision making, because the data will be available at the end of the EX stage, which happens during the same processor cycle. This problem leads to an unavoidable need to insert a nop between these two instructions. On the other hand, the hazard detection hardware is capable of detecting the need for data being in the EX/MEM pipeline register at the inputs of comparator's exclusive OR gates. Also, the forwarding unit is capable of supplying data from EX/MEM pipeline register to the comparator.

If the Branch Assumption button is off, the simulator inserts no nops after any branch instruction. This is also the only mode when two consecutive branch instructions will cause a program run time crash. Namely, if a branch instruction b_2 closely follows another branch instruction b_1 (supposing b_2 is a useful instruction that should be executed regardless whether b_1 has been taken or not), such program structure would have a difficult to predict behaviour, depending on whether b_1 has been taken or not.

Example

Consider the following fragment of a MIPS program

```
...
100 bne $1, $2, 10
104 beq $3, $4, -10
108 NOP
112 add
...
```

If the `bne` instruction has been not taken and `beq` instruction has been taken, the program will resume execution from the line 72. If both `bne` and `beq` instructions have been taken, then the program will execute the instruction on the line 148 first and then it will jump to the line 112.

7.3.1. The Summary Table

The table below describes the behaviour of the particular MIPS instructions in the case when the Forwarding button is on.

Instruction	Data Read	Data Written	Comment
R-type	From RF at the end of the stage 2 (ID)	Into register file at the start of the stage 5 (WB)	
Branch	From RF at the end of the stage 2 (ID)	If taken, the target address written into PC in the stage 2 (ID)	The target address computed in the stage 2 (ID)
Lw	From RF at the end of the stage 2 (ID)	Into register file at the start of the stage 5 (WB)	Data read from the main memory in the stage 4 (MEM)
Sw	From RF at the end of the stage 2 (ID)	Into the main memory in the stage 4 (MEM)	

There are two situations when the distance between an instruction i_2 following another instruction i_1 and needing its result, has to be at least one instruction to avoid a hazard to happen. These two situations happen when:

- The instruction i_1 is a `lw` instruction, and the instruction i_2 is an R-type, I-type, or memory type instruction, and
- The instruction i_1 is an R-type or I-type instruction and the instruction i_2 is a branch instruction.

By making the distance between i_1 and i_2 being at least one, we allow forwarding to make up.

If a branch instruction follows a `lw` instruction and needs its result, the distance between these two instructions has to be at least two. Otherwise a hazard will occur. Also, the content of a memory location directly following a branch instruction will be fetched and unconditionally executed. If the memory location contains an invalid op code, this will lead to a program crash.

Let ba be the address of a branch instruction ta be the target address, and os be branch word offset. Since sn the stage number in which the PC is updated is two, the branch word offset is computed in the following way:

$$os = (ta - 8 - ba) / 4.$$

7.3.2. Branch Assumption On

If Branch Assumption button is on, it means that a ***branch not taken*** assumption is made. So, an instruction following a branch instruction enters the pipeline. Moreover, if a branch is taken and the PC is updated, the instruction being in the IF stage will be flushed. The instruction flushed will be not executed and thus not counted.

Let ba be the address of a branch instruction ta be the target address, and os be branch word offset. Since sn the stage number in which the PC is updated is two, the branch word offset is computed in the following way:

$$os = (ta - 8 - ba) / 4.$$

8. How to Use the MIPS Pipeline Simulator

Store a MIPS assembly program prepared for running by MIPS Pipeline Simulator in your private directory.

We have a command line interface to MIPS Pipeline Simulator, so you need to run it from a UNIX prompt in a shell window. To enable the various applications required, first type

```
> need spim
```

You may wish to add the “need mpsim” command to your .cshrc file so that it is run automatically.

To start MIPS Pipeline Simulator type:

```
> mpsim
```

To execute a program, perform the following procedure:

1. Click on Clear Instructions button to reset the simulator.
2. Load your program using Load button (browse to your program if needed).
3. Choose a data_path operation mode by asserting appropriate mode buttons.
4. Execute your program either stepwise (using the Step button repetitively), or continuously (Clicking on the Run button).

It is advisable to step through your program first, to see how it behaves, and then to run it.