MIPS Assembler and Simulator

Reference Manual
Last Updated, 22nd April 2004

Xavier Perséguers

Swiss Federal Institute of Technology
xavier.perseguers@epfl.ch

Preface

MIPS Assembler and Simulator is a tool for converting assembly source code into machine code in either hexadecimal or binary output format. A simulation interface is also integrated.

The course ArchOrd, Processor Design 4th semester, from E. Sanchez and P. Ienne gave me the basis of the MIPS assembly language. We had to create a processor in VHDL and write a few programs to test it. As the work to convert from assembly code to machine code is somehow boring and slow to perform, I preferred starting to code a tool to do this job for me.

You may say: “Well, programmers have all the same ideas: if they have a task for the following week, they prefer spending the week coding a tool that will do their job in half an hour than doing it by hand...”

You’re right, but the difference is that I have yet a tool that has been and will be used by many colleagues for the same task. Next year, when 1st year students will have to do the same work, they will appreciate this tool.

Disclaimer

I programmed this tool for my own use. A few students used it for their own project. But I cannot be responsible of any damage or lost of time due to this tool. You use it at your own risk. If someone finds bugs, please do not hesitate to send me a mail.

This tool is distributed as *freeware*. If you find it is really useful, I would be very happy if you sent me an e-mail to let me know how you dealt with it :-)

Have a great time
Xavier Perséguers

Marly, July 2002

Users Comments

Switzerland

It's been a very exciting and enriching experience to follow the development of the MIPS Assembler. It's aimed not only to provide an assembly compiler for the MIPS, but also to help students understand the underlining architecture.

Xavier Perséguers has brought the power of several tools into one easy to use and consistent interface. I'm convinced that a lot of students will be grateful to his work, as they will discover the new dimension of assembly programming and computer design. *Alok Menghrajani, Swiss Federal Institute of Technology*

India

I have begun to use the MIPS Assembler since few weeks for my course assignments at IIT Delhi. While using the Assembler I found it was a complete tool for decently advanced MIPS programming. With the syntax checking option also included and the color coding of the assembly language it was fairly easy to debug the programs written. The insert options to include assembly of high level programming constructs was really good. I shall be using it for the present semester too so I shall be grateful if you would update me when the next version will be available. *Gaurav Bhatnagar, Indian Institute of Technology Delhi*

Contents

1 | MIPS Architecture

1.1 History: Processor architectures

1.1.1 1980: CISC

CISC stands for ***C**omplex **I**nstruction **S**et **C**omputer*.

- CISC processors try to minimize required memory used for storing instructions by increasing their complexity ;
- High-level instructions performing more than one operation at once (eg: storing multiple registers in memory) ;
- Instructions often take their operands directly from the memory ;
- Lots of sophistical addressing modes (eg: indirect, double indirect, indexed, post-incremented, . . .) ;
- Done with micro-code: each instruction is interpreted with a micro-code program ;
- Typical processors: Digital VAX, Motorola MC 68000, Intel 8086, Pentium.

1.1.2 1990: RISC

RISC stands for ***R**educed **I**nstruction **S**et **C**omputer*.

- In 1990, the hole between memory and processor has been accentuated ;
- Solutions:

  No micro-code, only hardware implementation ;

  High number of registers used for intermediate results and program variables ;

  Use of cache memories in order to speed up repetitive data access.
- Use of parallelism with a *pipeline* ;
- Need of a simple instruction set ;
- Typical processors: MIPS, Sun SPARC.

1.1.3 Historical Perspective on Units of Memory

There have been architectures that have used nearly every imaginable word sizes from 6-bit bytes to 9-bit bytes, and word sizes ranging from 12 bits to 48 bits. There are even a few architectures that have no fixed word size at all (such as the CM-2) or word sizes that can be specified by the operating system at runtime.

Over the years, however, most architectures have converged on 8-bit bytes and 32-bit longwords. An 8-bit byte is a good match for the ASCII character set (which has some popular extensions that require 8 bits), and a 32-bit word has been, at least until recently, large enough for most practical purposes.

1.2 Format of the MIPS Instructions

All instructions are encoded with 32 bits. MIPS instructions are grouped in three categories or *formats*. Each instruction contains all needed information to be executed. Always using a 32 bit representation allows a pipeline being easy to implement, unlike in x86 architectures such as Pentium.

1.2.1 Formats

I-Format ***(Data transfer, branch format)***

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

J-Format ***(Jump instruction format)***

op	address
6 bits	26 bits

R-Format ***(Arithmetic instruction format)***

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

1.3 Hazards

A hazard occurs whenever the pipeline tries to access a resource that is not yet available or after a branch instruction. Take a look at the following example:

```
addi $t0, $zero, 10
add $t1, $zero, $t0
```

The second instruction needs the value of the register $t0 when decoding the instruction (second stage of the 5-stages pipeline). The problem is that the result ($t0 = 10) has not yet been written back to the register when decoding the second instruction. This means that the processor will have a wrong value (in fact the previous value) for the register $t0 and the result will be wrong.

The different types of hazards are:

Structure hazard: Some combinations of operations are prohibited due to a given hardware implementation.

Data hazard: The result of an operation needs a previous result that is not yet available.

Control hazard: A branch instruction is executed with a delay. In the 5-stages pipeline, the delay is two (2) clock cycles. This means that the two instructions following a branch instruction will always be executed.

1.4 Types of Data Hazard

1.4.1 Read after Write (RAW)

An instruction tries to read a register or the contents of the memory after an instruction that writes to this resource.

Example

```
add $t1, $zero, $zero
addi $s0, $zero, 0xFF
sub $t0, $s0, $t1
```

1.4.2 Write after Read (WAR)

An instruction writes a value to a register or to the memory after an instruction that reads this resource. The MIPS pipeline's structure does not allow this hazard to occur.

1.4.3 Write after Write (WAW)

An instruction writes a value to a register or to the memory after an instruction that writes to this resource.

Example

```
add $t1, $zero, $zero
addi $t1, $zero, 0xA
```

1.4.4 Solving Hazards

Hazard may be solved either by Stalls or by Forwarding or even both.

1.5 Stalls

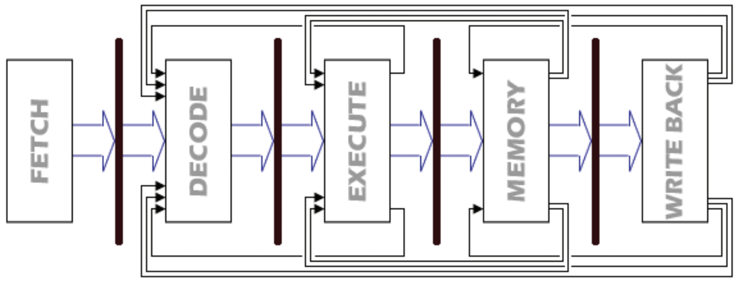
Stalls is used by the processor when an instruction needs a result that has been calculated but is not yet available as it has not been written back to the registers or to the memory. It is also needed with control hazards if the two following instructions do not have to be executed if the branch (conditional or unconditional) has to be taken.

Forwarding option is activated: The forwarding solves all kind of hazards. Standard stalls are useless. Consequently if stalls option is activated, stalls will be used to automatically insert 2 bubbles after each conditional and unconditional branch instruction. This way, control hazard will be automatically be solved.

Forwarding option is deactivated: Stalls needs to solve all kind of hazards. If an instruction is in the Fetch stage and the processor detects a data hazard, it will stop the pipeline until all sources registers are available.

1.6 Forwarding

Forwarding allows the processor to take a result that has not yet be rewritten to the registers or the memory as a "source" register for a following instruction that needs it to be executed. MIPS Assembler implements all kinds of forwarding:



Please keep in mind that the forwarding from the stage Write Back and the stage Decode is *always available* even if you deactivate the forwarding option.

1.7 MIPS Assembler Instructions Set

Note: Instructions with a star (*) are pseudo-instructions. This means that the instruction is not part of the MIPS instructions set. The pseudo-instruction will be converted to real MIPS instruction at assembly time.

1.7.1 Arithmetic and Logical Instructions

Instruction	Example	Meaning
add	`add $s1, $s2, $s3`	$s1 = $s2 + $s3
add w/o overflow	`addu $s1, $s2, $s3`	$s1 = $s2 + $s3
add immediate	`addi $s1, $s2, 10`	$s1 = $s2 + 10
add imm. w/o overflow	`addiu $s1, $s2, 10`	$s1 = $s2 + 10
AND	`and $s1, $s2, $s3`	$s1 = $s2 ∧ $s3
AND immediate	`andi $s1, $s2, 0xFD`	$s1 = $s2 ∧ 0xFD
NOR	`nor $s1, $s2, $s3`	$s1 = ¬ ($s2 ∨ $s3)
NOT*	`not $s1, $s2`	$s1 = ¬ $s2
OR	`or $s1, $s2, $s3`	$s1 = $s2 ∨ $s3
OR immediate	`ori $s1, $s2, 0xFD`	$s1 = $s2 ∧ 0xFD
shift left logical	`sll $s1, $s2, 4`	$s1 = $s2 << 4
shift left logical var.	`sllv $s1, $s2, $s3`	$s1 = $s2 << $s3
shift right logical	`srl $s1, $s2, 4`	$s1 = $s2 >> 4
shift right logical var.	`srlv $s1, $s2, $s3`	$s1 = $s2 >> $s3
subtract	`sub $s1, $s2, $s3`	$s1 = $s2 − $s3
subtract w/o overflow	`subu $s1, $s2, $s3`	$s1 = $s2 − $s3
subtract immediate*	`subi $s1, $s2, 10`	$s1 = $s2 − 10
exclusive OR	`xor $s1, $s2, $s3`	$s1 = $s2 ⊕ $s3
exclusive OR imm.	`xori $s1, $s2, 0xFD`	$s1 = $s2 ⊕ 0xFD

1.7.2 Comparison Instructions

Instruction	Example	Meaning
set on less than	`slt $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0
set on less than unsgn.	`sltu $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0
set on less than imm.	`slti $s1, $s2, 10`	if ($s2 < 10) $s1 = 1; else $s1 = 0
slt immediate unsgn.	`sltiu $s1, $s2, 10`	if ($s2 < 10) $s1 = 1; else $s1 = 0

1.7.3 Conditional Branch Instructions

Instruction	Example	Meaning
branch on equal	`beq $s1, $s2, L`	if (`$s1` == `$s2`) go to `L`
branch on not equal	`bne $s1, $s2, L`	if (`$s1` != `$s2`) go to `L`
branch on greater than equal zero	`bgez $s1, L`	if (`$s1` $\geq$ 0) go to `L`
branch on greater than zero	`bgtz $s1, L`	if (`$s1` $>$ 0) go to `L`
branch on less than equal zero	`blez $s1, L`	if (`$s1` $\leq$ 0) go to `L`
br. on less than zero	`bltz $s1, L`	if (`$s1` $<$ 0) go to `L`
branch on greater than equal zero and link	`bgezal $s1, L`	if (`$s1` $\geq$ 0) `$ra` = PC + 4; go to `L`
branch on less than zero and link	`bltzal $s1, L`	if (`$s1` $<$ 0) `$ra` = PC + 4; go to `L`

1.7.4 Constant-Manipulating Instructions

Instruction	Example	Meaning
load upper immediate	`lui $s1, 0xFD`	`$s1` = `0xFD` << 16
load immediate*	`li $s1, 0xFD`	`$s1` = `0xFD`

1.7.5 Jump Instructions

Instruction	Example	Meaning
jump	`j 0x2500`	go to addr `10,000`
jump and link	`jal 0x2500`	`$ra` = PC + 4; go to addr `10,000`
jump and link register	`jalr $s1, $s2`	`$s2` = PC + 4; go to `$s1`
jump register	`jr $ra`	go to `$ra`

1.7.6 Special Instruction

Instruction	Example	Meaning
no operation	`nop`	*do nothing*

The `nop` instruction is typically used with pipelined simulations when a bubble needs to be inserted after another instruction.

1.7.7 Data transfer Instructions

Instruction	Example	Meaning
load byte (sign-extended)	`lb $s1, 96($s2)`	$s1 = Memory[$s2 + 96]
load unsigned byte	`lbu $s1, 96($s2)`	$s1 = Memory[$s2 + 96]
load halfword (sign-extended)	`lh $s1, 96($s2)`	$s1 = Memory[$s2 + 96]
load unsigned halfword	`lhu $s1, 96($s2)`	$s1 = Memory[$s2 + 96]
load word	`lw $s1, 96($s2)`	$s1 = Memory[$s2 + 96]
store byte	`sb $s1, 96($s2)`	Memory[$s2 + 96] = $s1
store halfword	`sh $s1, 96($s2)`	Memory[$s2 + 96] = $s1
store word	`sw $s1, 96($s2)`	Memory[$s2 + 96] = $s1

1.7.8 Multiplication and Division Instructions

Instruction	Example	Meaning
multiplication	`mult $s1, $s2`	$hi = HIGH($s1 × $s2) $lo = LOW($s1 × $s2)
multiplication w/o overflow	`multu $s1, $s2`	$hi = HIGH($s1 × $s2) $lo = LOW($s1 × $s2)
division	`div $s1, $s2`	$hi = $s1 ÷ $s2 $hi = $s1 mod $s2
division unsigned	`divu $s1, $s2`	$hi = $s1 ÷ $s2 $hi = $s1 mod $s2
move from high	`mfhi $s1`	$s1 = $hi
move from low	`mflo $s1`	$s1 = $lo
move to high	`mthi $s1`	$hi = $s1
move to low	`mtlo $s1`	$lo = $s1

1.7.9 Exception and Interrupt Instructions

Instruction	Example	Meaning
syscall	`syscall` *`R, fmt, code`*	execute the system call with given parameters (*see* Tables 1.1 and 1.2)
break	`break` *`code`*	cause exception *`code`*

System calls may be used if you want to export your MIPS program as an executable (*see* § 2.9). **These instructions cannot be used when performing a pipelined simulation!**

1.7.10 Instruction for the Loader

Instruction	Example	Meaning
.start	`.start` *`address`*	the program will be loaded at position *`address`* in memory

Service	Code	Arguments	Meaning
print_int	1	$*R* = integer	print the content of register *R* as an integer
print_char	2	$*R* = character (value = byte)	print the ASCII character whose code is equal to the content of register *R*
read_int	4	$*R* stores the integer	read an integer from the standard input
read_char	5	$*R* stores the character in a byte	read a character from the standard input and store its ASCII code

Table 1.1: Syscall Services

Format	Description
0	common used value when service is not `print_int`
1	signed decimal integer
2	unsigned decimal integer
3	unsigned hexadecimal integer, using "abcdef"
4	unsigned hexadecimal integer, using "ABCDEF"
width = 0	
5	signed decimal integer
6	unsigned decimal integer
7	unsigned hexadecimal integer, using "abcdef"
8	unsigned hexadecimal integer, using "ABCDEF"
width = 4 (zero padded)	
9	signed decimal integer
10	unsigned decimal integer
11	unsigned hexadecimal integer, using "abcdef"
12	unsigned hexadecimal integer, using "ABCDEF"
width = 8 (zero padded)	
13	signed decimal integer
14	unsigned decimal integer
15	unsigned hexadecimal integer, using "abcdef"
16	unsigned hexadecimal integer, using "ABCDEF"

Table 1.2: Formatting numbers with `print_int`

1.8 MIPS Registers

The following table summarizes the MIPS register convention.

Name	Register number	Usage
`$zero`	0	the constant value 0
`$at`	1	reserved for the assembler
`$v0`–`$v1`	2–3	values for results and expression evaluation
`$a0`–`$a3`	4–7	arguments
`$t0`–`$t7`	8–15	temporaries
`$s0`–`$s7`	16–23	saved
`$t8`–`$t9`	24–25	more temporaries
`$k0`–`$k1`	26–27	reserved for the operating system
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

1.9 Miscellaneous

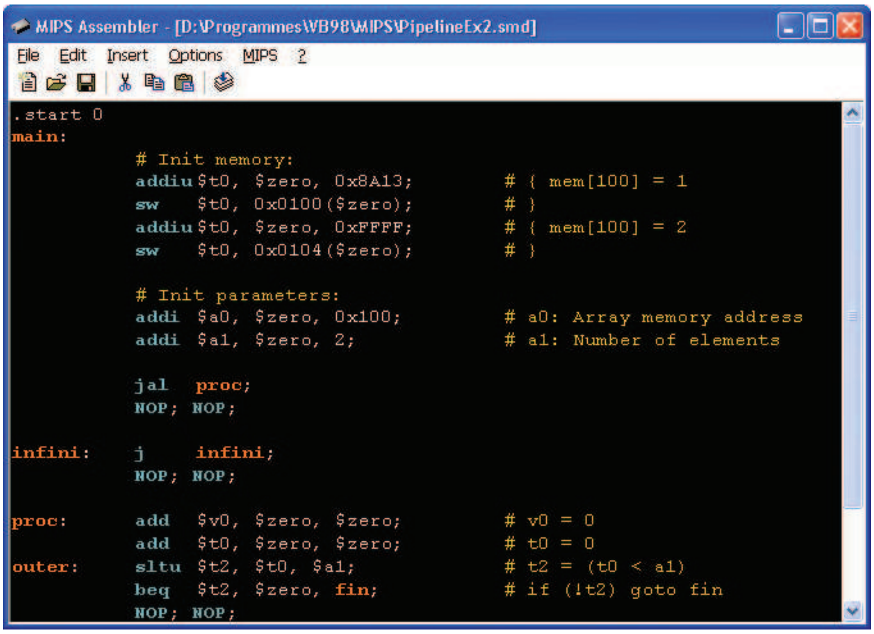
MIPS Simulator processor has a 128 words (32 bits) data memory, that is 512 bytes. The instruction memory is “unlimited”. MIPS executables (§ 2.9) have a 4 KB data memory.

1.10 Stack Pointer

The stack pointer `$sp` is preset to point to the end of the memory. This way, the stack grows from the end of the memory to the beginning.

2 Using MIPS Assembler and Simulator

2.1 Main/Assembly code window



This window is composed of 6 menus, a toolbar and an assembly code panel. The name of the current file is displayed in the title bar.

Assembly code panel

The colors for the code may be modified with the Configuration dialog (Menu Options). The code is not colored during the edition but at each time you assemble the code or if you choose the item Colorize Assembly in the Menu MIPS.

2.1.1 Menu File

New: This menu item creates a new assembly document

Open. . . : This menu item displays an Open dialog box to let you choose the assembly file you want to open.

Save: This menu item saves the current assembly source code. If your code has not already an associated file name, displays a Save dialog box.

Save As. . . : This menu item displays a Save dialog box and then save the current assembly source code using the file name you typed. Use this item when you want to save the modification in a new file instead of overwriting the previous version.

Print. . . : This menu item prints the source code. Be aware that the background color will not be printed. So if you work with white text on a black background, your text will be invisible!

Recent Files: These menu items provide a quick access to the most recent files you have worked with. To change the number of recent files displayed in this menu, go to the Configuration dialog box and select the Environment tab.

Quit: Quits the MIPS Assembler software.

2.1.2 Menu Edit

Cut: This menu item copies the contents of the selection to the clipboard and then deletes the selection. Does not preserve the formatting (font, colors, . . .) when copying

Copy: This menu item copies the contents of the selection to the clipboard. Does not preserve the formatting (font, colors, . . .) when copying. To preserve the formatting, use the following menu entry.

Copy as RTF: This menu item copies the contents of the selection to the clipboard while preserving the formatting (font, colors, . . .).

Paste: This menu item replaces the contents of the selection with the contents of the clipboard.

Select all: This menu item selects the whole source code.

Find. . . : This menu item displays a search dialog box.

Find Next: This menu item finds the next occurrence of the text you search.

Indent: This menu item indents the selected source code. That is, it adds a tab at the beginning of each line of code.

Outdent: This menu item outdents the selected source code. That is, it removes a tab from the beginning of each line of code, when possible.

Go To. . . : This menu item lets you navigate easily and quickly to a given line of code.

2.1.3 Menu Insert

This menu shows you templates categories on the first menu level and the associated templates at the second menu level. You may create your own templates with the Templates Editor tool. There is only two limitations: at most 10 categories and. . . your imagination!

More information may be found in chapter 4.

2.1.4 Menu Options

Hexadecimal machine code: This menu item lets you set the format of the instructions in the Machine Code window. If selected, hexadecimal notation will be used, otherwise the instructions will be shown in binary. This option is independent from the format of the machine code exportation.

Semicolon need after instruction: This option lets you specify whether you want that semicolon are needed after each instruction. It is useful when opening old source file where semicolons were needed. You may write more than one instruction on each line of text, even if the semicolon need is deactivated.

Configuration. . . : This menu item displays you the Configuration dialog box.

2.1.5 Menu MIPS

Check Syntax: This menu item checks that your assembly code is correct. If not, it displays a dialog box indicating what it did not understand.

Colorize Assembly: This menu item colorizes your assembly code. Your code is automatically colorized each time you assemble it. The colors for the code may be modified with the Configuration dialog (Menu Options).

Assembly: This menu item combines both previous actions (Check Syntax and Colorize Assembly) and then opens the Machine Code window to let you simulate your program.

2.1.6 Menu Help

Help: This menu item displays the help file.

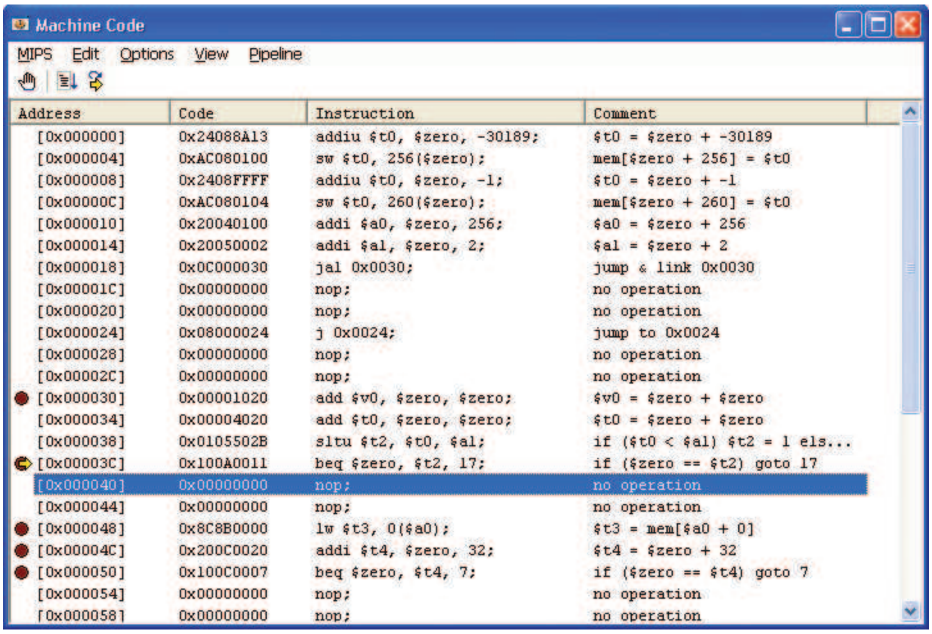
References: This menu item displays references. Click here for online references.

Templates Editor: This menu item opens the Templates Editor tool that let you create and modify templates (*see* chapter 4).

MIPS Assembler on the Web: This menu item takes you to the MIPS Assembler web site.

About. . . : This menu item displays an About dialog box.

2.2 Machine code window



2.2.1 Menu MIPS

Reset: This menu item lets you reset the current simulation. Registers are reinitialized, PC is reset to point to the first instruction (may not be `0x00000` if you did use the `.start` command) and the memory is cleared.

Execute Program: This menu item lets you execute a given number of instructions at once. Please note that the simulation will stop if a breakpoint is detected.

Next Instruction: This menu item executes one instruction (identified by the address stored in the PC). Use this button to perform a step-by-step simulation.

Toggle Breakpoint: This menu item adds or removes a breakpoint in front of the selected instruction. Use breakpoints to stop the simulation before executing the corresponding instruction.

Clear All Breakpoints: This menu item removes all breakpoints you may have set in the machine code.

2.2.2 Menu Edit

Copy Machine Code: This menu item exports the machine code to the clipboard. To change the exportation preferences, go to the Configuration dialog box and select the Machine Code Export tab.

2.2.3 Menu Options

Detect uninitialized registers: This menu item allows you to get a warning if your code reads a register without having setting it to an initial value.

2.2.4 Menu View

Registers / Memory: This menu item opens the Registers / Memory window.

Pipeline: This menu item opens one of the Pipeline window (5-Stages, 5-Stage Detailed), according to the selected pipelined simulation.

2.2.5 Menu Pipeline

No Pipeline: This is the standard mode simulation, without any pipeline. Each instruction is completely executed at each step of the simulation.

5-Stages: This menu item selects the 5-stage pipeline simulation.

Options > Stalls: This option activates / deactivates the use of stalls during the simulation.

Options > Forwarding: This option activates / deactivates the use of forwarding during the simulation.

Please note that this is the unique menu without check marks! As there are icons in front of the menu items, I preferred setting the caption to a bold font when the corresponding item was selected.

Special Behavior

Forwarding solves all hazards and no instruction needs stalls anymore, excepted branch instructions. Consequently if stalls option is activated while forwarding option is also activated, stalls will be used to automatically insert 2 bubbles after each conditional and unconditional branch instruction.

Options Summary

	Forwarding / Stalls (OFF)	Forwarding / Stalls (ON)
Stalls (ON)	If forwarding would be needed, the pipeline stops (ie. processor inserts a bubble) before decoding the instruction.	All-way forwarding. The processor automatically inserts two (2) bubbles after each branch instruction.
Forwarding (ON)	All-way forwarding. The two (2) instructions following a branch instructions are executed before the PC is modified.	All-way forwarding. The processor automatically inserts two (2) bubbles after each branch instruction.

The forwarding from the stage Write Back and the stage Decode is *always available*.

If both options are deactivated, no hazard is solved and you can be pretty sure your program will produce strange results. You might use this mode for school exercises.

2.3 Pipelines

2.3.1 5-Stages

This is the 5-stage pipeline overview. The five pipeline steps

1. Fetch
2. Decode
3. Execute
4. Memory
5. Write Back

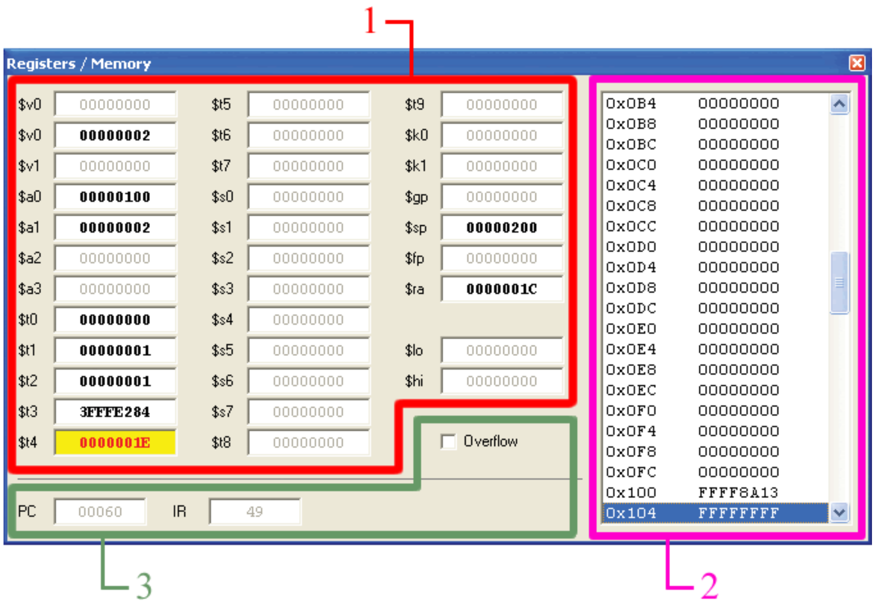
of the "execution" of a single instruction are shown.

Explanation

Bubbles instructions like NOPs the user typed will be shown as is in the pipeline. Bubbles instructions generated by the processor (stalls option is activated) will be shown as `---` (as if the pipeline stage does not contain any instruction).

The violet lines show you where a forwarding way was used (forwarding option is activated).

2.4 Registers/Memory window



This window is composed of three main parts:

1. The registers
2. The memory
3. PC and IR value / Overflow flag

Uninitialized registers are disabled. If you try to access their contents you may be notified that the register was not initialized (see Detect uninitialized registers in the menu Options of the Machine Code window). The last modified register is displayed in red on a yellow background. You may change the contents of a register by clicking on it *(left click for initialized registers, right click for uninitialized ones).*

The list shows you the contents of the memory. The last modified memory entry is highlighted. You may change an entry of the memory by double-clicking it in the list.

2.5 Types of Files

MIPS Assembler uses text files. Any text file containing assembly code may be used as a source file for the assembler. However MIPS Assembler is best dealing with three types of files:

.smd Default extension for MIPS assembly sources. “smd” stands for SimulMips Data, as it is the extension D. Baumann used in his SimulMips software. This is the software we had to use for our processor project ;

.asm Pseudo-binary source file. This is one of the greatest feature of MIPS Assembler. It allows you to type pseudo-binary instructions in any text editor and get the corresponding assembly code when you open it in MIPS Assembler.

.exe Executable file. This is the second greatest feature of MIPS Assembler. It allows you to desassemble a MIPS executable and get the corresponding assembly code when you open it in MIPS Assembler. Author's name is extracted. Please note that only MIPS executable may be open!

2.6 New Document's Template

MIPS Assembler may automatically load a template when creating new assembly files. For instance you might want to add custom headers to each of your programs. Then the best way is to add this header in the template file.

2.6.1 Customizing the Template

The template file is located in the MIPS Assembler install directory and is named `template.smd`. You may type any text but there is two meta-variable you may include:

@@AUTHOR@@ will be replaced at creation-time by the name MIPS Assembler saved as the default author's name (*see* § 2.7) ;

@@DATE@@ will be replaced at creation-time by the current date using european format `dd.mm.yyyy`.

2.7 Author's Name

Each time you export your assembly code as a LaTeX file (*.tex) or when you create an executable, MIPS Assembler asks you for the programmer's name. The value you provide will be saved and taken as the default value next time the dialog box is shown. If you want to override this setting, you may:

1. Add in a comment a line of type `# Author:` *`YOUR NAME`* (*see* § 2.6.1). This way, MIPS Assembler will not ask you to provide a name ;
2. Edit the value `Name` in the registry: `HKEY_CURRENT_USER\Software\VB and VBA Program Settings\MIPS Assembler\User`.

2.8 Decoding Pseudo-Binary Sources

You may have, as exercise, to decode a few instruction written in binary. Suppose you are asked to decode `0x2008000A`. The only work to do, if you do not want to decode it manually, is to open a text editor, such as notepad, type the instruction as you got it, either in hexadecimal format (starting with `0x`) or in binary format, using only `0` and `1` digits. Put one instruction per line. You may add comment as if you were in MIPS Assembler. Save the file using a `.asm` extension. Now open it in MIPS Assembler and go drink a cup of coffee while other students are still trying to decode the operation field...

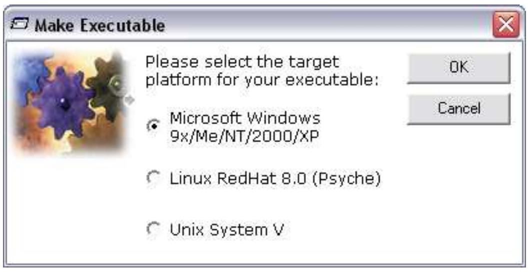
Note: It is a good idea to do this task a few times manually. If you get this exercise in a test (and that might be the case), you'll not have your laptop running MIPS Assembler on your desk!

MIPS Assembler is able to decode full programs written in assembly. It will even replace branch and jump offsets with smart labels, allowing you to better understand the assembly code.

You may export your own assembly code as pseudo-binary instructions with the menu "Edit" of the Machine Code window. Export configuration may be changed using the configuration dialog in the menu "Options" of the main MIPS Assembler window.

2.9 Exporting as Executable

Once you have assembled your MIPS program and you are happy with it, you may create an executable for any platform:



The executable file is actually an emulator written in C for having a small executable with the binary MIPS assembly code written at the end of it. This lets you create an executable for the platform you prefer! The name of the author is encoded in the executable file. Do not try to change it, it would require too much work. . . :-)

Please include a break instruction as you would include a `return` or `exit` statement in a C program.

2.9.1 Example

Source File multiples.smd

```
# ---------------------------------------------------------------
# Program:  Multiples
# Date:     07.03.2003
# Author:   Xavier Perséguers
# ---------------------------------------------------------------

li      $t6, 0x2A          # store the ASCII character '*'
li      $t7, 0x3D          # store the ASCII character '='
li      $t8, 0x20          # store the space character
li      $t9, 0xA           # store the new line character

syscall $a0, 0, 4          # Read an integer into $t0
li      $v0, 0

# ---- Start of the loop ---------------------
# Counter: $t0
# Start:   0
# End:     12
li      $t0, 0
li      $t1, 12
addi    $t1, $t1, 1
Loop:
    syscall $t0, 9, 1      # Print $t0
    syscall $t8, 0, 2
    syscall $t6, 0, 2      # Print *
    syscall $a0, 9, 1      # Print $a0
    syscall $t8, 0, 2
    syscall $t7, 0, 2      # Print =
    syscall $v0, 9, 1      # Print $t0 * $a0
    syscall $t9, 0, 2      # New line

    add     $v0, $v0, $a0  # calculate the next value

addi    $t0, $t0, 1
bne     $t0, $t1, Loop
# ---- End of the loop -----------------------

break       0
```

Execution in a DOS Window

```
C:\Programmes\VB98\MIPS>multiples
6
  0 *   6 =   0
  1 *   6 =   6
  2 *   6 =  12
  3 *   6 =  18
  4 *   6 =  24
  5 *   6 =  30
  6 *   6 =  36
  7 *   6 =  42
  8 *   6 =  48
```

```
  9 *   6 =  54
 10 *   6 =  60
 11 *   6 =  66
 12 *   6 =  72

C:\Programmes\VB98\MIPS>
```

2.9.2 Supporting other platforms

If you would like to let MIPS Assembler support another platform, please download the emulator source code from the website:

`http://icwww.epfl.ch/~persegue/mips/src/emul.tar.gz`

and compile it using the associated `Makefile` or with any C compiler. You then have to copy the executable to the `\bin` subdirectory of the MIPS Assembler install directory and edit the file `mipsexport.ini` in the MIPS Assembler install directory.

```
[Exportation]
NumberOfOS=3

OS1=Microsoft &Windows 9x/Me/NT/2000/XP
Binary1=bin\emul_windows.bin
Extension1=.exe

OS2=&Linux RedHat 8.0 (Psyche)
Binary2=bin\emul_redhat8.bin
Extension2=

OS3=&Unix System V
Binary3=bin\emul_unix.bin
Extension3=
```

Figure 2.1: Default `mipsexport.ini` file

3 | MIPS Assembly Syntax

Unlike other assembly languages, instructions *may* be terminated with a semicolon. You are allowed to write more than one instruction per line, even if you do not terminate instructions with a semicolon. The optional use of semicolon is only to allow you opening any type of MIPS source files (eg: SimulMips).

3.1 Grammar

3.1.1 General

assembly_code ::= [".start" number] {instruction}

number ::= decimal_number | hexadecimal_number

decimal_number ::= –32,768. . . 32,767 | 0. . . 65,535 *(range may be more bounded for some instructions)*

hexadecimal_number ::= 0x0000. . . 0xFFFF *(leaded zeros may be omitted; range may be more bounded for some instructions)*

instruction ::= "add" register "," register "," register
| "addi" register "," register "," number
| "lui" register "," number
| "lw" register "," number "(" register ")"
| "sw" register "," number "(" register ")"
| "beq" register "," register "," (number | label)
| "j" number | label
| . . . (*see* § 1.7)

register ::= register_name | register_number

register_name ::= "$zero" | "$at" | . . . | "$ra"

register_number ::= "$0" | "$1" | . . . | "$31"

label ::= *see* § 3.1.2

3.1.2 Labels

Labels may be used in place of offsets / addresses for branch and jump instructions. Labels are composed of any number of letters from A to Z, digits from 0 to 9, dot (.) and underscore (_) symbols. Labels must start with a letter. When defining a label, you have to add a colon at the end of the name of the label.

Grammar

label *(definition)* ::= letter {letter | digit | special_character} ":"

label *(in instruction)* ::= letter {letter | digit | special_character}

letter ::= "a" | "b" | "c" | ... | "z" | "A" | "B" | "C" | ... | "Z"

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

special_character ::= "." | "_"

3.2 Comments

3.2.1 Embedded in Source Code

Comments are introduced with the pound sign (#) and terminated with a new line. This means that comments are valid until the end of the line. They may be written after an instruction.

3.2.2 During Simulation

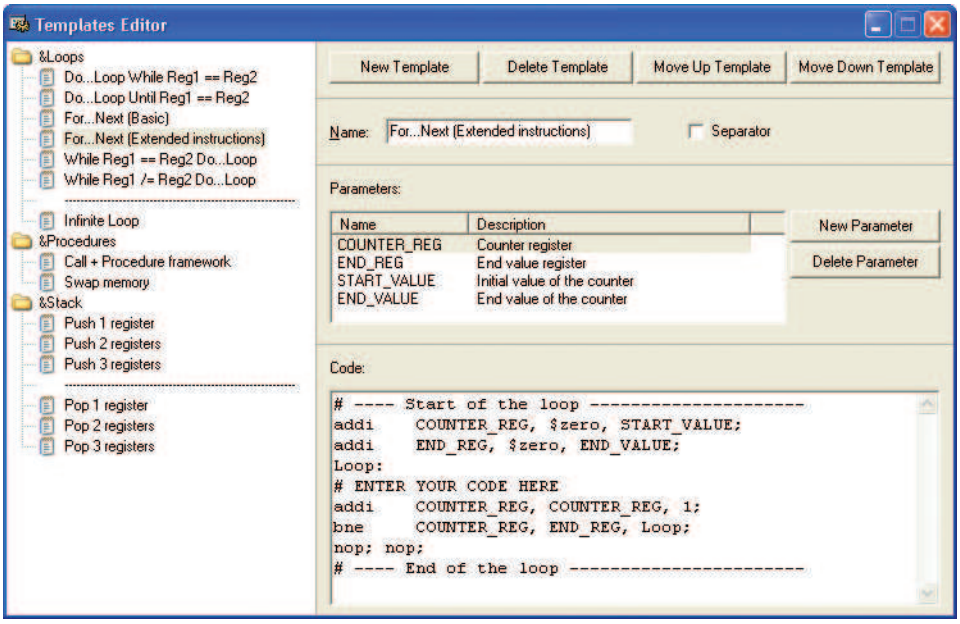
MIPS Assembler automatically generates comments when either displaying the machine code or exporting it. These comments may sometimes differ from what you would have thought. Indeed MIPS Assembler tries to find the best comment for describing a given instruction.

3.2.3 Special Interpretations

Assembly	Interpretation	Comment
sll $zero, $zero, 0	nop	*no operation*
srl $zero, $zero, 0	nop	*no operation*
sllv $zero, $zero, $zero	nop	*no operation*
srlv $zero, $zero, $zero	nop	*no operation*
add $zero, $zero, $zero	nop	*no operation*
addu $zero, $zero, $zero	nop	*no operation*
sub $zero, $zero, $zero	nop	*no operation*
subu $zero, $zero, $zero	nop	*no operation*
		end on next page...

Assembly	Interpretation	Comment
addi $zero, $zero, 0	nop	*no operation*
addiu $zero, $zero, 0	nop	*no operation*
subi $zero, $zero, 0	nop	*no operation*
add R1, $zero, R2	add R1, $zero, R2	R1 = R2
add R1, R2, $zero	add R1, R2, $zero	R1 = R2
addu R1, $zero, R2	addu R1, $zero, R2	R1 = R2
addu R1, R2, $zero	addu R1, R2, $zero	R1 = R2
addi R1, $zero, number	addi R1, $zero, number	R1 = number
addiu R1, $zero, number	addiu R1, $zero, number	R1 = number
sub R1, R2, $zero	sub R1, R2, $zero	R1 = R2
sub R1, $zero, R2	sub R1, $zero, R2	R1 = – R2
subu R1, R2, $zero	subu R1, R2, $zero	R1 = R2
subu R1, $zero, R2	subu R1, $zero, R2	R1 = – R2
subi R1, $zero, number	addi R1, $zero, -number	R1 = – number
not R1, R2	not R1, R2	R1 = ¬ R2
nor R1, R2, R2	not R1, R2	R1 = ¬ R2

4 Templates Editor



4.1 Categories

New Category. This command button lets you create a new category. A category is symbolized with a folder icon. An input box will ask you to provide a name for the category. The category will be added after all the existing ones.

Delete Category. This command button lets you delete an existing category. **Be careful:** all templates associated to the corresponding category will be lost!

Move Up Category. This command button lets you move up a category. This means that if you select for instance the category *&Procedures* and you click on this button, the order of category won't be

1. &Loops
2. &Procedures

3. &Stack

anymore, but

1. &Procedures
2. &Loops
3. &Stack

Move Down Category. This command button lets you move down a category.

4.2 Templates

New Template. This command button lets you create a new template in the current category. A category is symbolized with a folder icon while a template with a file icon. An input box will ask you to provide a name for the template. The template will be added after all the existing ones in the current category.

Delete Template. This command button lets you delete an existing template.

Move Up Template. This command button lets you move up a template as you may have done for categories.

Move Down Template. This command button lets you move down a template.

New Parameter. This command button lets you create a new parameter for the current template. Parameters allow you to write generic assembly code and customize the template.

Delete Parameter. This command button lets you delete the selected parameter of the current template.

4.3 Default Templates

As you may have noticed, a few templates are provided with the MIPS Assembler distribution. Templates may be used whenever common recurring assembly code needs to be written. This allows you to only give a few parameters to get a full, valid block of assembly code. I created a way for organizing the templates, this is why you will find the *category*.

The purpose of this section is to show you how you may use existing templates.

4.3.1 Loops

Do...Loop While Reg1 == Reg2

Description: Loop structure with condition at the end of the loop. The loop is executed as long as the contents of two given registers are equal.

Parameters: REGISTER_1 and REGISTER_2. When asked, provide the name of these two registers (eg.: `$t0` and `$t1`)

Do...Loop Until Reg1 == Reg2

Description: Loop structure with condition at the end of the loop. The loop is executed until as the contents of two given registers are equal, that is, as long as they are not equal.

Parameters: REGISTER_1 and REGISTER_2. When asked, provide the name of these two registers (eg.: `$t0` and `$t1`).

For...Next

Description: For... Loop structure.

Parameters: COUNTER_REG: Name of the register holding the current loop's counter (this is the *i* in `for(i=1;i<=10;i++)`).
END_REG: Name of the register holding the value the counter needs to go to.
START_VALUE: The initial counter value (this is 1 in `for(i=1;i<=10;i++)`).
END_VALUE: The end counter value (this is 10 in `for(i=1;i<=10;i++)`).

While Reg1 == Reg2 Do...Loop

Description: Loop structure with condition at the beginning of the loop. The loop is executed as long as the contents of two given registers are equal.

Parameters: REGISTER_1 and REGISTER_2. When asked, provide the name of these two registers (eg.: `$t0` and `$t1`).

While Reg1 /= Reg2 Do...Loop

Description: Loop structure with condition at the beginning of the loop. The loop is executed as long as the contents of two given registers are not equal.

Parameters: REGISTER_1 and REGISTER_2. When asked, provide the name of these two registers (eg.: `$t0` and `$t1`).

Infinite Loop

Description: Default is a jump to the same instruction. Useful to stop the execution of a program without getting errors.

4.3.2 Procedures

Call + Procedure framework

Description: Code for calling and defining a procedure (or function) in your assembly code. Comments allow you to easily find how filling “blanks”.

Parameters: PROCEDURE_NAME: Name of your procedure/function.

Swap memory

Description: Procedure that may be used for easily swapping the contents of two memory addresses.

Usage: Store both addresses in `$a0` and `$a1` before jumping (`jal`) to `Swap`.

4.3.3 Stack

Push 1 register

Description: Code for reserving 4 bytes on the stack and saving the contents of a given register.

Parameters: REGISTER: Name of the register to push on the stack.

Push 2 registers

Description: Code for reserving 8 bytes on the stack and saving the contents of two given registers.

Parameters: REGISTER_1: Name of the 1st register to push on the stack.
REGISTER_2: Name of the 2nd register to push on the stack.

Push 3 registers

Description: Code for reserving 12 bytes on the stack and saving the contents of three given registers.

Parameters: REGISTER_1: Name of the 1st register to push on the stack.
REGISTER_2: Name of the 2nd register to push on the stack.
REGISTER_3: Name of the 3rd register to push on the stack.

Pop 1 register

Description: Reverse code as "Push 1 register": gets 4 bytes from the stack and saves it to a given register and then free space on the stack.

Parameters: REGISTER: Name of the register to be used for popping the stack.

Pop 2 registers

Description: Reverse code as "Push 2 registers": gets 8 bytes from the stack and saves them to two given registers and then free space on the stack.

Parameters: REGISTER_1: Name of the 1st register to be used for popping the stack.
REGISTER_2: Name of the 2nd register to be used for popping the stack.

Note: Registers are taken back in the reverse order as they were pushed on the stack. That is, REGISTER_*n* for operation "Push" should be REGISTER_*n* for operation "Pop".

Pop 3 registers

Description: Reverse code as "Push 3 registers": gets 12 bytes from the stack and saves them to three given registers and then free space on the stack.

Parameters: REGISTER_1: Name of the 1st register to be used for popping the stack.
REGISTER_2: Name of the 2nd register to be used for popping the stack.
REGISTER_3: Name of the 3rd register to be used for popping the stack.

Note: Registers are taken back in the reverse order as they were pushed on the stack. That is, REGISTER_*n* for operation "Push" should be REGISTER_*n* for operation "Pop".

4.3.4 Initialization

Register = Value

Description: Code for initializing the contents of a register with an immediate value.

Parameters: REGISTER: Name of the register to initialize.
VALUE: Immediate value (decimal or hexadecimal integer).

Register = Register

Description: Code for initializing the contents of a register with the contents of another register.

Parameters: REGISTER_1: Name of the register to initialize.
REGISTER_2: Name of the register to be used as source.

4.3.5 Your own Templates

4.4 Creating Templates

Templates are organized into categories, which will be shown as first-level menu entries in MIPS Assembler. You may create at most 10 different categories (including those already given).

A template is no more than pure assembly code with embedded parameters. The convention for parameters is to write them in capital letters. Each time you need a parameter to be included in your code, just write its name (and create an entry for it in the parameter list). When you insert the template in a source code, MIPS Assembler will ask you to provide a value for each parameter.

5 | Programming MIPS

5.1 From High-Level Programming to Assembly

This section will focus on some basis on how you may translate a source code in C, Java, Ada or any other high-level programming language into its assembly representation.

5.1.1 Arithmetic and Logical Operations

Example 1

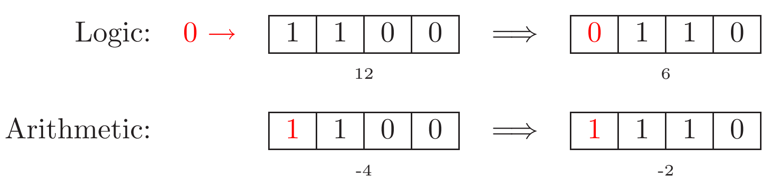
```c
void main() {
    int a = 2;
    int b = 10;

    a += b++;
    b = b | a;
}
```

We may choose `$s1` and `$s2` for variables a and b.

```asm
addi    $s1, $zero, 2     # a = 2;
addi    $s2, $zero, 10    # b = 10;
add     $s1, $s1, $s2     # a = a + b;
addi    $s2, $s2, 1       # b = b + 1;
or      $s2, $s2, $s1     # b = b | a;
```

Shifts on Signed Numbers

The figure below shows you the difference between logical and arithmetic shifts.



5.1.2 Accessing Memory

There is only two types of operations for accessing memory. The first one lets you read data from memory and store it in a register, while the other does the inverse, it lets you save the content of a register in the memory. A memory address must be aligned on four bytes.

Loading data from memory

`lw destination, offset(base);`

with:

destination:	destination register;
base:	register containing the base address;
offset:	positive number specifying an offset from the base address.

Storing data to memory

`sw source, offset(base);`

with:

source:	source register;
base:	register containing the base address;
offset:	positive number specifying an offset from the base address.

5.1.3 Exercises

Exercise 1: Write MIPS instruction for $g = h + A[8]$ where A is an array of 100 words, g and h are associated with `$s1`, `$s2`. Base address of A is stored in `$s3`.

Solution

```
lw  $t0, 32($s3)
add $s1, $s2, $t0
```

Exercise 2: $A[12] = h + A[8]$.

Solution

```
lw  $t0, 32($s3)
add $t0, $s2, $t0
sw  $t0, 48($s3)
```

Exercise 3: $g = h + A[i]$ where i is associated with `$s4`.

Solution

```
add $t1, $s4, $s4
add $t1, $t1, $t1
add $t1, $t1, $s3
lw  $t0, 0($t1)
add $s1, $s2, $t0
```

Exercise 4:

```
for (i = 0; i < j; i++)
    A[i] = B[i] + c;
```

where A, B, are array of words, the base address of A is in `$a0`, the base address of B is in `$a1` and $(i, j, k) =$ (`$s0`, `$s1`, `$s2`).

Solution

```
Loop:   slt $t0, $s0, $s1
        beq $t0, $zero, Exit
        add $t1, $s0, $s0
        add $t1, $t1, $t1
        add $a1, $a1, $t1
        lw  $t2, 0($a1)
        add $t2, $t2, $s2
        add $a0, $a0, $t1
        sw  $t2, 0($a0)
        addi $s0, $s0, 1
        j   Loop
Exit:
```

5.1.4 If-Then-Else

Will will now focus on how you may translate *if-then-else* structures into assembly. First of all keep in mind that there is no unique way of doing it. Let consider a "max" function that takes two arguments, a and b, and stores in c the greatest of its arguments. We will use `$a0`, `$a1` and `$v0` for representing the variables a, b and c.

```
if (a > b)
    c = a;
else
    c = b;
```

The idea is to store the result of the condition in a temporary register and then branch either to the TRUE or the FALSE part. We cannot test and branch in one instruction as there is no conditional branch for the operation *greater than* using two registers as argument. In addition, there is no *greater than* instruction but a *less than* — actually a *set less than* — instruction which stores either 1 (if true) or 0 (if false) in the destination register. Hence we have to change the code to

```
if (b < a)
    c = a;
else
    c = b;
```

Solution

```
    slt $t0, $a1, $a0       # $t0 = (b < a ? 1 : 0)
    beq $t0, $zero, L1      # false => go to L1
    add $v0, $zero, $a0     # true => c = a
```

```
    j   L2                  # do not evaluate the false part!
L1: add $v0, $zero, $a1     # c = b
L2:
```

5.1.5 Exercises

Exercise 1: How would you translate the code below?

```
if (a > b || (b < c && a == b)) {
    a = b * c;              // we suppose b and c are small
    c = 0;
}
```

Solution

```
    slt $t0, $a1, $a0       # $t0 = (b < a ? 1 : 0)
    slt $t1, $a1, $a2       # $t1 = (b < c ? 1 : 0)
    sub $t2, $a0, $a1       # $t2 = (a != b)
    nor $t2, $t2, $t2       # $t2 = (a == b)
    and $t1, $t1, $t2       # $t1 = (b < c && a == b)
    or  $t0, $t0, $t1       # $t0 = (a > b || (b < c && a == b))
    beq $t0, $zero, L1      # condition is false
    mult $a1, $a2
    mflo $a0                # a = LOW(b * c) as b, c are small
    add $a2, $zero, $zero   # c = 0
L1:
```

Exercise 2: You are asked to optimize the previous solution for faster execution using *short circuit*; that is not evaluating part of the expression if you already know the result. Eg: `if ((A && B) || C)`. In this condition, you do not need to evaluate `B` nor `C` if `A` is false...

Solution

```
    slt $t0, $a1, $a0       # $t0 = (b < a ? 1 : 0)
    bne $t0, $zero, L1      # true => short circuit to True part
    slt $t1, $a1, $a2       # $t1 = (b < c ? 1 : 0)
    beq $t1, $zero, L2      # false => short circuit: exit if
    bne $a0, $a1, L2        # false => condition is false
L1: mult $a1, $a2
    mflo $a0                # a = LOW(b * c) as b, c are small
    add $a2, $zero, $zero   # c = 0
L2:
```

5.2 Representing programs

Just as groups of bits can be used to represent numbers, they can also be used to represent instructions for a computer to perform. Unlike the two's complement notation for integers, which is a standard representation used

Figure 5.1: MIPS R2000 Instruction Formats

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Register	op	reg1	reg2	des	shift	funct
Immediate	op	reg1	reg2	16-bit constant		
Jump	op	26-bit constant				

by nearly all computers, the representation of instructions, and even the set of instructions, varies widely from one type of computer to another.

The MIPS architecture, which is the focus in this document, uses a relatively simple and straightforward representation. Each instruction is exactly 32 bits in length, and consists of several bit fields, as depicted in figure 5.1.

The first six bits (reading from the left, or high-order bits) of each instruction are called the *op* field. The op field determines whether the instruction is a *register*, *immediate*, or *jump* instruction, and how the rest of the instruction should be interpreted. Depending on what the op is, parts of the rest of the instruction may represent the names of registers, constant memory addresses, 16-bit integers, or other additional qualifiers for the op.

If the op field is 0, then the instruction is a register instruction, which generally perform an arithmetic or logical operation. The *funct* field specifies the operation to perform, while the *reg1* and *reg2* represent the registers to use as operands, and the *des* field represents the register in which to store the result. For example, the 32-bit hexadecimal number `0x02918020` represents, in the MIPS instruction set, the operation of adding the contents of registers 20 and 17 (`$s4` and `$s1`) and placing the result in register 16 (`$s0`).

Field	op	reg1	reg2	des	shift	funct
Width	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Values	0	20	17	16	0	`add`
Binary	000000	10100	10001	10000	00000	100000

if the op field is not 0, then the instruction may be either an *immediate* or *jump* instruction, depending on the value of the op field.

5.2.1 Exercises

Exercise 1: Using the Appendix A in *Computer Organization & Design*, perform the following encoding:

1. `and $a0, $t2, $t3` (answer in binary);

2. `addi $s1, $s4, -2` (answer in hexadecimal).

Solutions

1. `00000001010010110010000000100100`;
2. `0x2291FFFE`.

Exercise 2: Using the Appendix A in *Computer Organization & Design*, perform the following decoding:

1. `00111000011000100000000100000000`;
2. `0x000A8902`.

Solutions

1. `xori $v0, $v1, 0x100`;
2. `srl $s1, $t2, 4`.

5.3 Test your MIPS knowledge

This section offers you another set of questions and problems but, this time, without solution.

Question 1: Please give three ways for multiplying the content of a register by 4. We assume the value is small enough to avoid overflow.

Question 2: What can you say about performance (CPU usage) with the solutions of the question 1?

A Examples

NOP's special instructions have been added to the examples to let you simulate them without problem when forwarding or stalls are deactivated. You may remove them or try to put useful instructions instead of losing clock cycles.

A.1 Multiplication

This computes the multiplication of two numbers stored in `mem[$a0]` and `mem[$a1]` using its mathematical definition:

$$a \cdot b = \sum_{1}^{a} b.$$

```
# Initialization

addi $s0, $zero, 6              # First Operand
addi $s1, $zero, 8              # Second Operand
addi $a0, $zero, 4              # $a0 = 0
addi $a1, $a0, 4                # $a1 = 4
addi $v0, $a1, 4                # $v0 = 8
sw $s0, 0($a0)                  # mem[$a0] = $s0
sw $s1, 0($a1)                  # mem[$a1] = $s1

jal Multiplication
nop nop

lw $t0, 0($v0)                  # $t0 = mem[$v0]

Eternity: beq $zero, $zero, Eternity
nop nop

Multiplication:
lw $s0, 0($a0)                  # $s0 = operand #1
lw $s1, 0($a1)                  # $s1 = operand #2
add $s2, $zero, $zero       # $s2 contains temp result
add $s3, $zero, $zero       # $s3 is the loop counter
Loop:
add $s2, $s2, $s1           # $s2 = $s2 + $s1
addi $s3, $s3, 1            # $s3 = $s3 + 1
beq $s3, $s0, End           # if $s3 = $s1 GOTO End
```

```
nop nop
beq $zero, $zero, Loop        # GOTO Loop
nop nop
End:
sw $s2, 0($v0)                # store result in mem[$v1]
jr $ra
nop nop
```

A.2 Swap

This swaps the contents of two values in the memory based on their addresses.

```
# Base Addresses

addi $a0, $zero, 0x10
addi $a1, $zero, 1

# Initialize the RAM

addi $t0, $zero, 1
sw $t0, 0($a0)
addi $t0, $zero, 2
sw $t0, 4($a0)
addi $t0, $zero, 3
sw $t0, 8($a0)
addi $t0, $zero, 4
sw $t0, 0xC($a0)

# Swap RAM v[1] and v[2]

jal Swap
nop nop
Eternity: beq $zero, $zero, Eternity
nop nop

Swap:
    add $v0, $a1, $a1         # $v0 = 2 * $a1
    add $v0, $v0, $v0         # $v0 = 4 * $a1
    add $v0, $a0, $v0
    lw $t7, 0($v0)
    lw $t8, 4($v0)
    sw $t8, 0($v0)
    sw $t7, 4($v0)
    jr $ra
    nop nop
```

References

[1] Daniel J. Ellard. *CS50 Discussion and Project Book, Assembly Language Programming*, BBN HARK Systems Corporation, 1995.

[2] Xiangyun Gong. *MIPS Tutorials*, University of Wellington, 2002.

[3] Martin Odersky. *Compilation course*, EPFL / Programming Methods Laboratory, 2002–2003.

[4] David A. Patterson and John L. Hennessy. *Computer Organization & Design, 2nd Edition*, Morgan Kaufmann Publishers, Inc., 1998.

[5] Eduardo Sanchez and Paolo Ienne. *Computer Architecture course, 3rd and 4th semester*, EPFL / Processor Architecture Laboratory, 2001–2002.

Index

Symbols

A

B

C

D

F

H

I

J

L