A TinyMIPS ISA simulator

This report explains the design and implementation of an ISA simulator named TinyMIPS. The ISA follows rules of MIPS architecture [1], for example naming rules [2] and addressing modes [3]. The ISA currently support 32-bit signed integers and 32-bit unsigned memory address. Each instruction is 32-bit long and main memory is 4byte aligned.

# System Implementation

Develop environment: Windows 8.1x64, Dreamweaver CS6

Runtime environment: Google Chrome v33 dev-m

Programming languages: HTML5/CSS, JavaScript

Extra libraries: jQuery 2.0.3, jCanvas

# Register Design

Currently 14 registers were implemented which include 8 GPRs and 6 SPRs. All registers were given a 5bit address in order to keep consistency with MIPS ISA format. The GPR are named “r0”, “r1” and so on, and “r0” is always zero which is used for move instruction and comparing to zero in branch operations. SPRs are:

* PC: program counter
* NP: npc which is used by branch and jump to indicate next instruction address
* IR: instruction register for instruction fetching
* LM: LMD for load operation
* HI/LO: used for 32-bit integer multiplication and division

Following table shows the address of each registers and since the address is 5bit we can add up to 32 registers by editing the configuration file (register.js).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r0 | 00000 | r1 | 00001 | r2 | 00010 | r3 | 00011 | r4 | 00100 | r5 | 00101 | r6 | 00110 |
| r7 | 00111 | pc | 01000 | np | 01001 | ir | 01010 | lm | 01011 | hi | 01100 | lo | 01101 |

# Opcode Design & Instruction Format

Each opcode is 6bit which is same to MIPS. Opcode and oprands are separated by single space, oprands are separated by ‘,’ and different instructions are separated by line break (\r). The following instructions (with binary code) were implemented to support programs with basic 32-bit signed integer arithmetic operations:

* **syscall** **000000** For interaction with OS and program termination,

In MIPS the parameter is saved in register but in this design a 5bit integer is used. “syscall 0” means end of program

* **add** **000001** Add two registers and save to one,

add r1,r2,r3 reg[r1]=reg[r2]+reg[r3]

* **addi 000010** Add register with 16bit immediate and save to one register

addi r1,r2,3 reg[r1]=reg[r2]+3. Move equivalent using addi:

addi r1,r2,0 means move r2 to r1 and addi r1,r0,3 means move 3 to r1

* **sub 000011** subtraction which is similar to add
* **subi 000100** immediate subtraction which is similar to addi
* **div 000101** divide two registers and move quotient to hi and reminder to lo

div r1,r2 reg[hi]=r1/r2 reg[lo]=r1%r2

* **mul 000110** multiply two registers and save the 64bit integer to hi and lo

mul r1,r2 reg[hi]=(r1\*r2) [63..32] reg[lo]= (r1\*r2)[31..0]

* **load 000111** load from address in register with a 16bit offset:

load r1,r2,10 reg[r1]=mem[reg[r2]+10]

* **lui 001000** load 16 bit immediate to upper of a register

lui r1,10 reg[r1]=10<<16

* **ori 001001** bit-or 16 bit immediate with lower of a register

ori r1,10 reg[r1]=reg[r1]|10, using lui and ori together, we can move a 32bit address to a register

* **store 001010** store the value in one register to address in another with 16bit offset:

store r1,r2,10 mem[reg[r2]+10]=reg[r1]

* **jez 001011** compare register with r0 and jump to 16bit immediate if equal:

jez r1,10 if r1==0 npc+=10<<2, since all instruction address ends with two 0, 16bit immediate can indicate 18bit offset. All following conditional jump are similar to this.

* **jnz 001100** jump not zero
* **jgz 001101** jump greater than zero
* **jgez 001110** jump greater or equal zero
* **jlz 001111** jump less than zero
* **jlez 010000** jump less or equal zero
* **jr 010001** unconditional jump to a 32bit address in register:

jr r1 npc=reg[r1]

* **j 010010** jump based on 26bit unsigned immediate:

j 123 npc=pc[31..28] concat 123<<2, shift 26bit immediate 2bit left and concatenate with first 4bit of program counter to get a 32bit address. Thus can jump 1/16 of all instruction memory addresses.

* **mfhi 010011** move hi to a register

mfhi r1 reg[r1]=reg[hi]

* **mflo 010100** move lo to a register, these two instructions can be used to retrieve results from MUL and DIV

mflo r1 reg[r1]=reg[lo]

# Non-pipeline Execution

“Step” button is used to trace the execution of each instruction. The changing of register and memory are displayed in corresponding tables. The “Run” button will execute all instructions till the end of instruction set (if there is an end). Click on “Nopip” button will give a new page showing the instructions statistics including clock cycle and count of each instruction, and calculation of CPI, MIPS and execution time with default CPU frequency 2.1GHz (which can by modified and update the MIPS and execution time instantly).

# Pipelining

The pipeline of this ISA has 5 stages: IF, ID, EX, ME, WB. After “Run” the program click “Pipe” to show the pipeline stages.

## Branch Delay

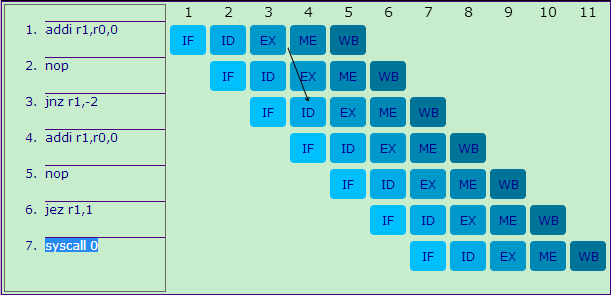
Branch predict taken was use to handle branch delay. The branch target instruction is fetched after the branch ID, if branch is taken the branch target instruction continue to ID, otherwise the branch+1 instruction is fetched and executed while target instruction is cancelled. Figure 1 shows the case that both branch is predict taken while the first branch is not taken and the second branch is taken. 

Figure 1: Predict taken

## RAW Hazards and Stall

RAW hazards are handled by forwarding. Figure 2 show a complex case with forwarding: (2) needs r1 from (1), (3) needs r2 from (2) and r1 from (1), and (4) needs r3 from (3) and r1 from (1).

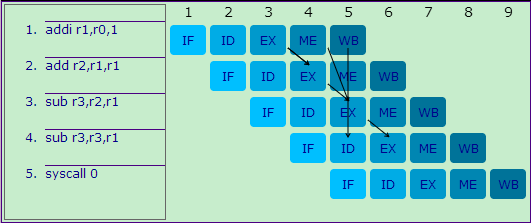


Figure 2: RAW hazards with forwarding

Some hazards cannot be handled by forwarding alone. Figure 3 show the pipeline of following instruction set. NOP was inserted to salve the data dependency problem with branch since branch need to calculate target address in ID. Indeed, every branch (conditional jump) has an arithmetic logic instruction ahead which produce the value used in the condition.

addi r1,r0,1

jez r1,-2

syscall 0

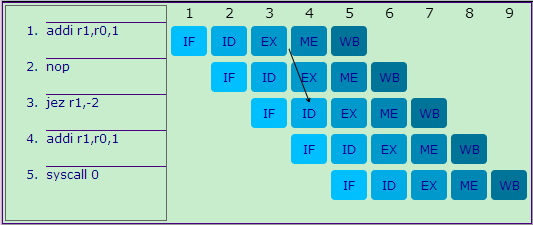


Figure 3: Insert NOP for branch

Another data hazard which is caused by “load” will have to stall since the data is only available after ME stage. Figure 4 shows the stall caused by load instruction

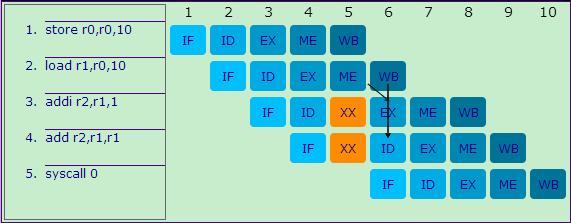


Figure 4: Stall caused by load

# References

[1] MIPS architecture, <http://en.wikipedia.org/wiki/MIPS_architecture>

[2] MIPS R3000 Instruction Set Summary, <http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>

[3] Summary of Addressing Modes in MIPS, <http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Mips/addr.html>