Computer Organization and Architecture

Introduction to the MIPS Processor

Prof. Pedro Frosi Rosa, Ph.D.

Faculty of Computing (FACOM)
Federal University of Uberlândia
Uberlândia, MG, Brazil
mehar.facom.ufu.br

Disclaimer

This presentation is a translation of the “Introduction à l’assembleur MIPS” originally developed by Professor Jean-Christophe Filliatre (http://www.lri.br/~filliatr).

Agenda

- MIPS – Concepts and Overview
  - MIPS Architecture and Assembly
  - MIPS simulator system calls
- MIPS Assembly
  - Directives
  - Labels
  - Instructions/Format
  - Memory/Stack
  - Functions
  - MIPS in a nutshell

MIPS: Intro

- MIPS - the target processor for this course
- Reading: MIPS on wikipedia
- Reference by regarding MIPS Processor

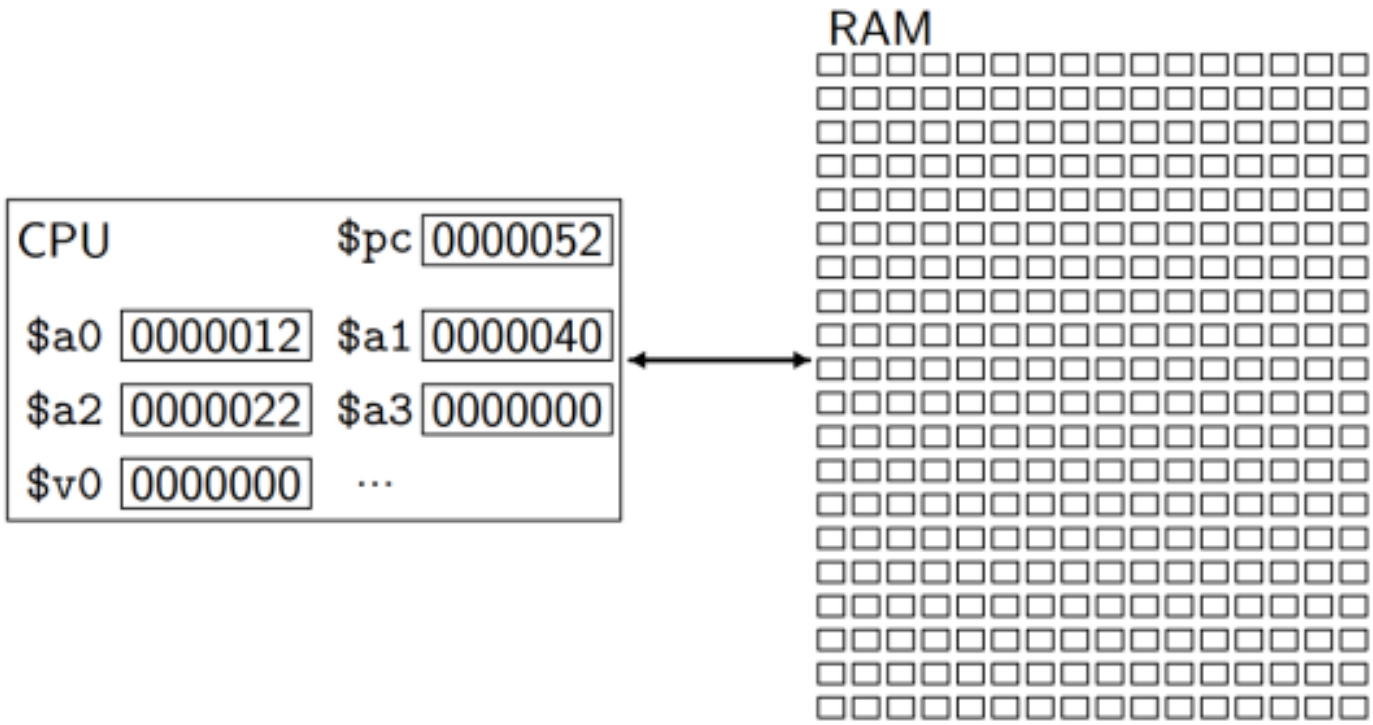
  Computer Organization & Design: The Hardware/Software Interface, Second Edition. By John Hennessy and David Patterson. Published by Morgan Kaufmann, 1997. ISBN 1-55860-428-6.

- MARS: MIPS Simulator

  http://courses.missouristate.edu/kenvollmar/mars/

  Note: The first important step is to go through Help !!!!

MIPS: Memory versus Register

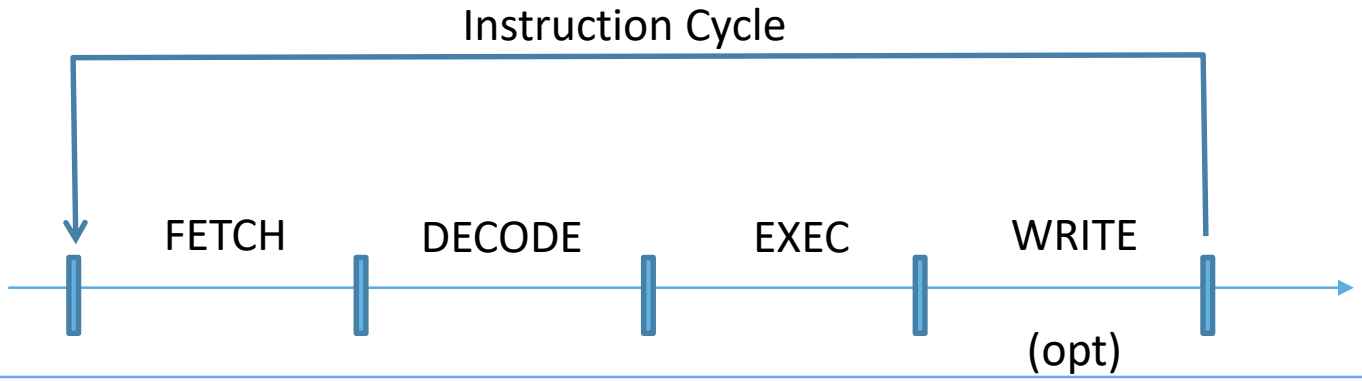


MIPS: Concepts and Reality

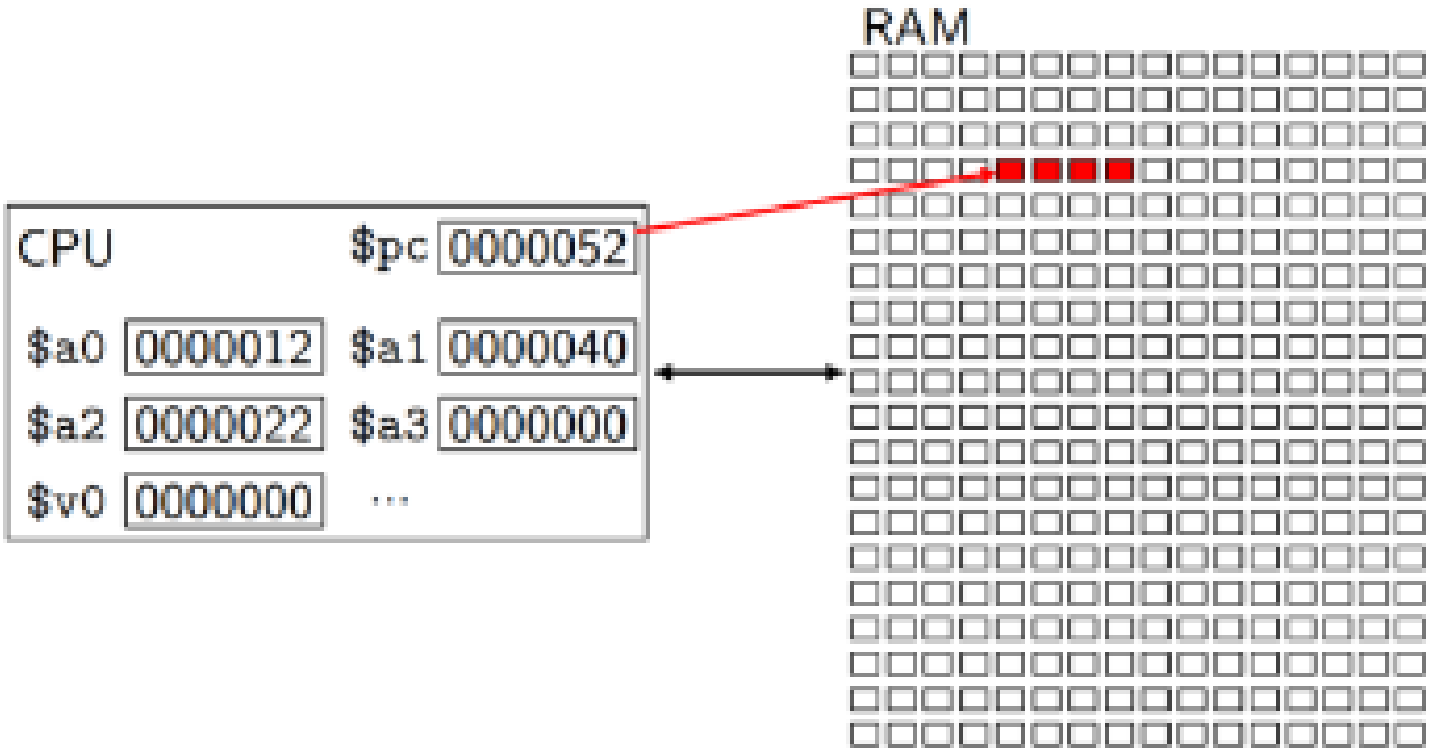
- The reality is a bit more complex
  - Potential presence of several (co)processors, some of them dedicated to floating point operations
  - One or more cache memories (replace politics, allocation and so on)
  - Virtual Memory is a reality (MMU)
  - Datapath
  - and many, many others aspects

The program execution principle

- The register PC ($pc) contains the address of the next instruction to be executed
- Memory read 4 (or 8) bytes addressed by PC (FETCH)
- The bitstring is parsed to find the instruction (DECODE)
- The decoded instruction is then executed (EXEC)
- The reached result is eventually write back (WRITE)
- The register PC is thus incremented to point out the next instruction



Example: Code and Assembly



Instruction	000000	00001	00010	0000000000001010
Decode	add	$a1	$a2	10
Meaning	$a1 <- $a2 + 10			

MIPS Architecture

- Registers (32) namely $r_0 - r_{31}$
  - r_0 contains allways zero
  - They can be referenced by other names, as:
    - zero, at, v0-v1, a0-a3, t0-t9, s0-s7, k0-k1, gp, sp, fp, ra
  - There are 3 types of instructions:
    - Transfer (memory ⟷ register)
    - Arithmetic – Fixed point (integer) or Floating point (real)
    - Jump (which modify PC)

MIPS Instructions

- Load Immediate (li) Load Unsigned Immediate (lui)

```
li $a0, 64          ; $a0 <- 64
lui $a0, 64         ; $a0 <- 64 * 2^16
```

- Move register content

```
move $a0, $a1               ; $a0 <- $a1
```

MIPS: Arithmetic

- Add(ition)

```
add $a0,$a1,$a2          ; a0 <- a1+a2
add $a1,$a1,$t3          ; a1 <- a1+t3
```

(same for sub, mul, div)

- Immediate Addition (only for add)

```
addi $a0,$a1,64          ; a0 <- a1+64
```

- Negation

```
neg $a0,$a1          ; a0 <- -a1
```

- Absolute

```
abs $a0, $a1             ; a0 <- |a1|
```

MIPS: Bitwise operators

- Logical negation (not $10110_2 = 01001_2$)

```
not $a0,$a1 ; a0<-not(a1)
```

- And Logic operator (and (10110_2 , 01001_2)=00000_2)

```
and $a0,$a1,$a2  ; a0<-and(a1,a2)
andi $a0,$a1,0x2f      ; a0<-and(a1, 101111₂)
```

- Or Logic operator (or(10110_2 , 01001_2)= 11111_2)

```
or $a0,$a1,$a2   ; a0<-or(a1,a2)
ori $a0,$a1,0x2f; a0 <- or (a1, 101111₂)
```

MIPS: Shift operators

- Logic shift left (zeroed insertion)

```
sll $a0,$a1,2      ; a0<-a1*(2^2)
sllv $a0,$a1,$a2   ; a0<-a1*2^a2
```

- Arithmetic Shift right (keep signal)

```
sra $a0,$a1,2      ; a0<-a1/(2^2)
```

- Logic shift right (zeroed insertion)

```
srl $a0,$a1,2
```

- Rotation

```
rol $a0,$a1,2
ror $a0,$a1,2
```

MIPS: Transfer (read)

- Read memory (a 32 bit word)

```
lw $a0, 64($a1)    ; a0 <- MEM[a1+64]
```

  - The targeted address is a 16-bit displacement relative to register (a1 in this case)
  - There are instructions for 8 or 16 readings signed or not (lb, lh, lbu, lhu)

MIPS: Transfer (store)

- Write memory (a 32 bit word)

```
sw $a0, 64($a1)          ; MEM[a1+64]<-a0
```

  - PS: take care about parameters ordering
  - The targeted address is a 16-bit displacement relative to register (a1 in this case)
  - There are instructions for 8 or 16 storings (sb, sh)

MIPS: Branch (jump)$_1$

- Conditional branch

```
beq $a0,$a1,label        ; if (a0==a1) the label
```

  - Variants (bne, blt, ble, bgt, bge)
  - Variants for zero (beqz, bnez, bltz, blez, bgtz, bgez)

MIPS: Branch (jump)$_2$

- Unconditional branch
  - For a label

```
j label
```

  - For a label, by linking current address in `$ra` register

```
Jal label     ; jump and link
```

  - For an address stored in a register

```
jr $a0
```

  - For an address stored in a register and keep the next instruction address in another register

```
jalr $a0,$a1
```

    - » PS: branch to the address stored in register `a0` and store the next instruction address in `a1`

MIPS: System Call

The hardware interface offers some services to abstract certains details beneath processos architecture (at a simulator they are services of the underlying operating system)

- The MIPS calls this kind of services a System Call (**syscall**)
- Service Code must be assigned to the **`v0`** register ($v0)
- Arguments must be assigned to the **`a0-a3`** registers ($a0-$a3)
- The result could be retrieve from the **`v0`** register ($v0), if any

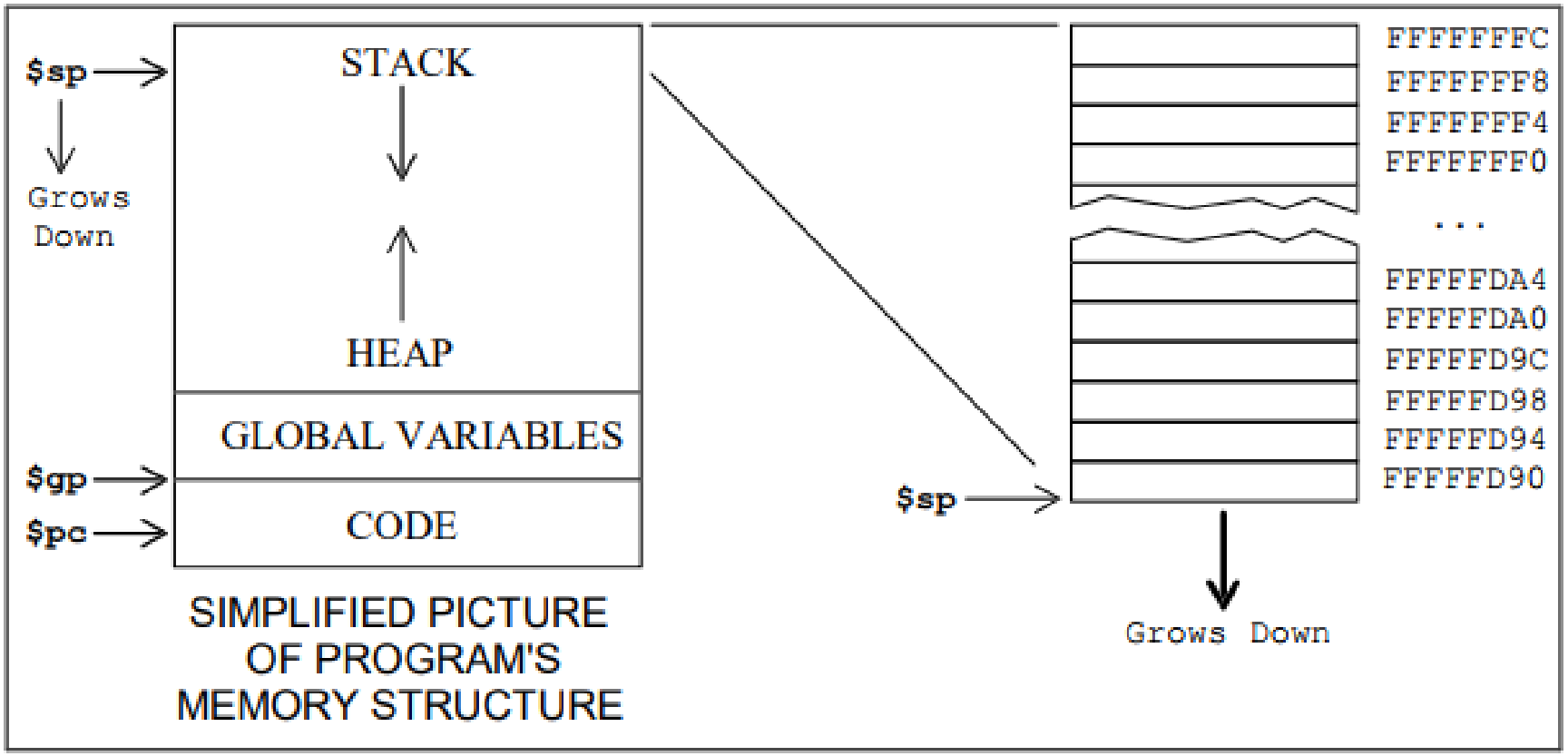
– Example, the service '**Print Integer**'

```
li $v0,1        ; Print Integer service code
li $a0,64       ; Value to be printed
syscall         ; call the simulator interface
```

MIPS Assembly

- Microcode programming is a hard task and it is not a comum activity
  - Microcode programming is sometimes referenced as machine language
- Assembly offers certain facilities, as:
  - Symbolic labels: symbols (words) to abstract memory addresses;
  - Global data allocation: global variables and heap memory utilization;
  - Pseudo instructions: these are simple assembly language instructions that do not have a direct machine language equivalent.

MIPS Memory Structure



MIPS Assembly Directives & Labels

`.text`

- Determines to the compiler where the program (code) reside

`.data`

- Tells assembler we're in the data segment (program variables)

`.global`

- Tells assembler a symbol is accessible outside file

`label:`

- Tells assembler the memory address it represents

MIPS: Hello World!

```
        .text
        .global main        ; main is accessible outside
main:   li    $v0,4         ; Print Character String
        la    $a0,hello     ; Character string address
        syscall             ; Do it
        li    $v0, 10       ; End of program
        syscall             ; Do it

        .data
hello:  .asciiz "Hello World!\n"

;PS: See other directives as .byte .space .word aso
```

MIPS and the Compiler

- The challenge for compilers (MIPS in particular) is to translate high level language to the instruction set, by regarding:
  - Control structure (comparisons, loops, exceptions aso);
  - Functions calls (procedures aso);
  - Complex data structures (matrix, arrays, objects aso);
  - Heap memory allocation

Calls to functions (1)

- Functions could be called randomly, e.g., they could be arbitrarily nested
  - Register could not be enough to store all the parameters, local variables and other local data to the calling
  - It is necessary to allocate memory for it

- Functions are nested/attended in a *last-in-first-out* base (stack)
- Revisit MIPS memory structure on page 20

Calls to functions (2)

- Suppose two functions *f* and *g*, where *f* calls *g*:
  - Function *f* is named **caller** and function *g* is named **callee**
  - Function *f* calls *g* as follows:

```
jal g          ; jump and link g
```

  - The instruction `jal` store in the register **`ra`**, the address of the next instruction
  - To return back, the caller (that is, the function *g*) must have as the last instruction:

```
jr $ra         ; jump register
```

  - But:
    - If *g* becomes a caller?
    - If *g* modifies some registers used before its calling? What about caller *f*?
- There must have a standard to make calls!!!!!

Calls conventions

- About registers:
  - at, k0, k1 are reserved to the operating systems
  - a0, …, a3 are used to pass parameters to the functions (others come through stack)
  - v0, v1 are used to return back calls results
  - t0, …, t9 are caller saved registers, caller must store them, if necessary, being that they are used typically to those data which do not need to survive after call
  - s0, …, s7 are called saved registers, callee must store them, if necessary, being that they are used typically to those data which do need to survive after call
  - sp,fp, ra, gp are, respectively, stack pointer, frame pointer, return address, and the pointer to the static data segment

Four steps calling process

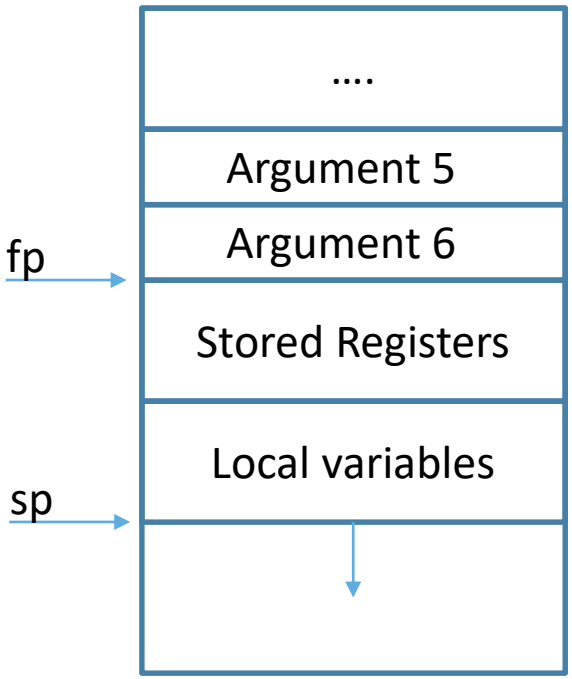
- They are:
  - (1) For the caller, before calling
  - (2) For the callee, at the beginning
  - (3) For the callee, before returning
  - (4) For the caller, after calling
- The steps are organized around a stack frame located between fp and sp register
- Some literatures reference stack frames as activation table.

Caller: before calling (step 1)

- Assigns registers a0-a3
  - If there are less than 4 parameters, leave remaining registers untouched
  - If there are more than 4 parameters, pass them through stack
- Save the registers t0-t9, as necessary, to further utilization
- Call the called function

```
jal callee
```

Callee: at the beginning (step 2)



(1) Establish your stack frame, for instance:

```
addi $sp,$sp-28
```

(2) Store fp and set it, for example:

```
sw $fp,24($sp)
addi $fp,$sp,24
```

(3) Store s0-s7 e ra

```
sw $fp,24($sp)
addi $fp,$sp,24
```

Importante: fp allows to reach the arguments and local variables, based on a fixed offset, no matter the stack state.

Callee: before returning (step 3)

(1) Set the function result on v0 and, if any, v1

(2) Restore the registers

(3) *Pop* of its stack frame

```
addi $sp, $sp, 28
```

(4) Return back

```
jr $ra
```

Caller: after calling

(1) Pop eventually the arguments 5, 6 aso (those which have been passed through stack)

(2) Restore *called-saved* registers

MIPS in a nutshell (1)

Instruction	Meaning
li $r0,C	r0 <- C
lui $r0,C	r0 <- 2^16*C
move $r0, $r1	r0 <- r1
add $r0, $r1, $r	r0 <- r1+r2
addi $r0, $r1, C	r0 <- r1+C
sub $r0, $r1, $r2	r0 <- r1-r2
div $r0, $r1, $r2	r0 <- r1/r2
div $r1, $r2	lo <- r1/r2, $hi <- r1 mod r2
mul $r0, $r1, $r2	r0 <- r1 * r2 (no overflow)
neg $r0, $r1	r0 <- -r1
slt $r0, $r1, $r2	r0 <-1 iff r1 < r2, r0 <- 0 otherwise
slti $r0, $r1, C	r0 <-1 iff r1 < C, r0 <- 0 otherwise
sle $r0, $r1, $r2	r0 <- 1 iff r1 <= r2, r0 <- 0 otherwise
seq $r0, $r1, $r2	r0 <- 1 iff r1 = r2, r0 <- 0 otherwise
sne $r0, $r1, $r2	r0 <- 1 iff r1 <> r2, r0 <- 0 otherwise

MIPS in a nutshell (2)

Instruction	Meaning
la $r0, adr	r0 <- adr
lw $r0, adr	r0 <- mem[adr]
sw $r0, adr	mem[adr] <- r0
beq $r0, $r1, label	Jump (pc=label) iff r0 = r1
beqz $r0, label	Jump iff r0 = 0
bgt $r0, $r1, label	Jump iff r0 > r1
bgtz $r0, label	Jump iff r0 > 0
beqzal $r0, label	Jump iff r0 = 0 and let ra <- pc + 1
bgtzal $r0, label	Jump iff r0 > 0 and let ra <- pc + 1
j label	Jump (unconditional)
jal label	Jump and let ra <- pc + 1
jr $r0	Jump r0
jalr $r0	Jump r0 and let ra <- pc + 1

MIPS: Registers calling convention

Name	Number	Use	Callee must preserve?
$zero	$0	C=0	N/A
$at	$1	Assemb temp	No
$v0-$v1	$2-$3	Values for function returns and expression evaluation	No
$a0-$a3	$4-$7	Function arguments	No
$t0-$t7	$8-$15	Temporaries	No
$s0-$s7	$16-$23	Saved temporaries	Yes
$t8-$t9	$24-$25	Temporaries	No
$k0-$k1	$26-$27	Reserved to OS kernel	N/A
$gp	$28	Global pointer	Yes
$sp	$29	Stack pointer	Yes
$fp	$30	Frame pointer	Yes
$ra	$31	Return address	N/A

Questions?

Thank you!

pfrosi@ufu.br

Prof. Pedro Frosi Rosa, Ph.D.

http://mehar.facom.ufu.br

Faculty of Computing (FACOM)

Federal University of Uberlândia (UFU)