CpE 313: Microprocessor Systems Design

Handout 04
MIPS Architecture

September 09, 2004
Shoukat Ali

shoukat@umr.edu

UMR UNIVERSITY OF MISSOURI-ROLLA
The Name. The Degree. The Difference.

Credits

- Mazin Yousif, Intel
- EECS @ Berkeley
- Prof. Jung-Min @ Virginia Tech

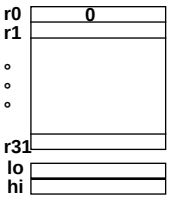
Introduction

- MIPS ISA
  - MIPS = **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
  - microprocessor architecture developed by MIPS Computer Systems Inc.
  - used by NEC, Nintendo, Silicon Graphics, Sony
  - one of the first RISC designs
  - design goal: maximize performance and minimize cost
  - 32-bit architecture
    - all ALU operations are conducted on 32-bit long words
      - 32-bit addition, subtraction, etc.

3

MIPS Overview – Load/Store Architecture

- 31 x 32-bit GPRs (R0=0)
- 32 x 32-bit FP regs OR 16 64-bit double precision FP registers
- HI, LO (special purpose registers)



food for thought

- what if the result of an integer operation is too big to fit into the 32 bits of a register?
- what if it is too small?
  - how does the CPU fill extra space?

- value of R0 is always ZERO
  - will discuss how this helps later
- just 32 registers? why not more? aren't "lotsa registers" better?
  - trade-off: more the registers, more the bits needed to address them. IN ADDITION, more the time needed to reach a register

4

MIPS Overview – Data Types & Addressing Modes

- data types
  - 8-bit bytes, 16-bit half words, and 32-bit words for integer data
  - 32-bit single precision and 64-bit double precision for FP data
- every operation modifies all 32-bits of the result register
- what if the CPU fetches only one byte into a register?
  - how do we fill up the extra register space?

- addressing modes
  - only provides immediate and displacement addr modes
    - e.g., add R1, R2, #30 … or … “load” R1, 100(R2)
  - “you said register indirect was also important”
    - achieved by setting displacement value to 0 in disp addr mode
  - what about absolute addressing mode?
    - achieved by using R0 in disp addr mode

5

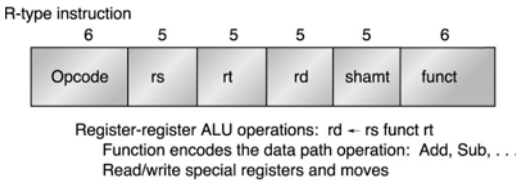
MIPS Overview – Instruction Format

- all instructions are 32 bits long!
  - easy to _____
- since only two addressing modes, they are encoded in the opcode
  - opcode ends in “i” for immediate mode
    - add versus addi
- MIPS requires different kinds of instruction formats for different kinds of instructions
  - R-type (register)
    - for register arithmetic instructions
  - I-type (immediate)
    - for data transfer instructions, conditional branches
    - instructions w/ immediate operands
  - J-type (jump)
    - for unconditional jump instructions

6

MIPS Arithmetic

- exactly three operands: all must be registers
  - no memory-memory or register-memory ALU operations
    - load/store architecture!!
- operand order is fixed
  - destination, source operand 1, source operand 2
  - example: **`add $1,$2,$4`**
    - convention: put a $ sign before the register name
  - example: **`add $1,$2,$zero (`**trick for **`mov)`**



- opcode: basic operation of the instruction
- rs: 1st reg source operand
- rt: 2nd reg source operand
- rd: reg destination operand
- shamt: shift amount
- funct: function code, selects specific variant of opcode

7

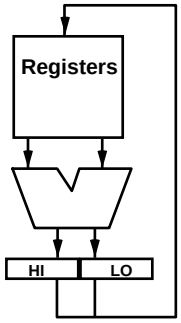
MIPS Arithmetic Instructions: Add/Subtract

Instruction	Example	Meaning	Comments
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions

- arithmetic operations can create results that are
  - too big to fit in 32 bits
    - called overflow
    - an exception must be raised by the CPU when overflow happens
  - too small to fit in 32 bits
    - need to sign or zero-extend
- why do we need unsigned add/subtract?

8

MIPS Arithmetic Instructions: Multiply/Divide



- HI and LO are two special 32-bit registers just used to hold multiply/divide results
  - why special? only four instructions can access HI,LO
- multiplying an m-bit number with an n-bit number:
  - result is m+n bits
  - how do we ensure that multiplying two #s gives a 32-bit number? (why 32-bit?)
    - by ensuring _____

Instruction	Example	Meaning	Comments
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu $2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

9

Some MIPS Logical Instructions

Instruction	*Example*	*Meaning*	*Comment*
and	**and $1,$2,$3**	**$1 = $2 & $3**	**3 reg. operands; Logical AND**
and immediate	**andi $1,$2,10**	**$1 = $2 & 10**	**Logical AND reg, constant**
shift left logical	**sll $1,$2,10**	**$1 = $2 << 10**	**Shift left by constant**
shift right logical	**srl $1,$2,10**	**$1 = $2 >> 10**	**Shift right by constant**
shift right arithm.	**sra $1,$2,10**	**$1 = $2 >> 10**	**Shift right (sign extend)**

no sign-extension for operations like andi!

10

MIPS Data Transfer Instructions

Instruction	*Comment*
SW 500(R4), R3	Store word
SH 502(R2), R3	Store half
SB 41(R3), R2	Store byte

Encodes: Loads and stores of bytes, half words, words, double words. All immediates **(rd ← rs op immediate)**

LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load halfword
LHU R1, 40(R3)	Load halfword unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned

why two types?

LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16)

note: Stores never sign/zero extend

Why do we need LUI?

11

LUI – Loading Large Constants

- want to load 32-bit constant into register $1
  - ISA only allows 16-bit constants
    - use load upper immediate (lui) instruction

constant = 0000 0000 0011 1101 0000 1001 0000 0000

load upper 16 bits

lui $1, 61 # 61_{10} = 0000 0000 0011 1101_2

contents of $1	0000 0000 0011 1101	0000 0000 0000 0000

add lower 16 bits, whose decimal value is 2304

addi $1, $1, 2304 # 2304_{10} = 0000 1001 0000 0000_2

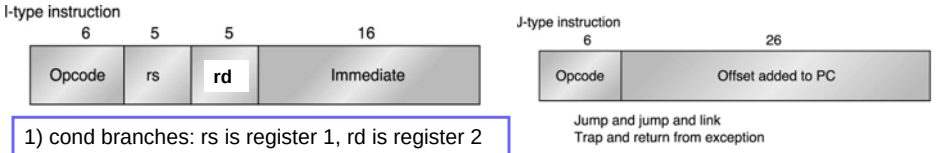
contents of $1	0000 0000 0011 1101	0000 1001 0000 0000

do you regret not having allowed 32-bit immediates?

12

MIPS Jump, Branch, Compare instructions

Instruction	*Example*	*Meaning*
branch on equal	beq $1,$2,100	if ($1 == $2) PC= PC+4+100
	Equal test; PC relative branch	
branch on not eq.	bne $1,$2,100	if ($1!= $2) PC = PC+4+100
	Not equal test; PC relative	
jump	j 10000	PC = 10000
	Jump to target address	
jump register	jr $3	PC = $3
	For procedure return, switch statement	
jump and link reg	jalr $2	$31 = PC + 4; PC = $2
	For procedure call	



1) cond branches: rs is register 1, rd is register 2
2) jumps: rs is register, immediate = rd = 0

13

Break!

14

MIPS Convention for Register Names

format: register number, name, usage

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(callee must save)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address (HW)

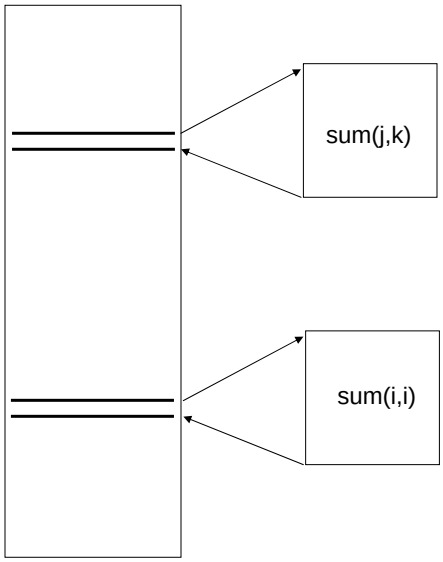
15

Procedure Calls and Returns

```
main() {
  int i,j,k,m;
 j = 30;
 k = 20;
 i = sum(j,k);
 /* other instructions */;
 m = sum(i,i);
}
int sum (int addend1, int addend2){
 int sum;
 sum = addend1 + addend2;
 return sum;
}
```

16

Procedures – Control Flow



issues

- procedures must be able to return to the "instruction after"
  - save return address in $ra
  - returning procedure just jumps to address in $ra
- procedure must know where to get its input
  - i.e., where to get j and k
  - caller prog puts j and k in registers $a0 and $a1
  - invoked procedure reads these "argument registers"
- prog must know where to find the result of procedure
  - when done, proc puts its results in reg $v0 and $v1

17

Procedures – Example

C

```
... sum(j,k);... /* j,k:$s0,$s1 */
}

int sum(int x, int y) {
    return x+y;
}
```

MIPS

```
address
1000 add  $a0,$s0,$zero  # x = j
1004 add  $a1,$s1,$zero  # y = k
1008 jalr sum #$ra=1012 and jump to sum
1012 add  $t0,$t0,$t1
1016 ...
2000 sum: add $v0,$a0,$a1
2004 jr $ra
```

18

Procedure Call Bookkeeping

- before jumping to procedure
  - put the arguments of procedure in registers $a0, …, $a3
  - save the return address in $ra

- while in the procedure
  - <refer to chalkboard>

- before returning from procedure
  - put the results in registers $v0 and $v1

19

Procedure Call Bookkeeping

- before jumping to procedure
  - put the arguments of procedure in registers $a0, …, $a3
  - save the return address in $ra

- while in the procedure
  - use registers $t0, …, $t7 for local variables
  - could also use $s0, …, $s7 if needed, but must restore $s0, …, $s7 to original values before returning to main program
    - this approach is known as _____
  - why not save restore $t0, …, $t7 as well?
    - $t0, …, $t7 are “caller saved”

- before returning from procedure
  - put the results in registers $v0 and $v1

20