ISA Part IV
Instruction Representation2
MIPS Assembly - Miscellaneous

MIPS Instructions (Quick Summary)

Name	Example	Comments
	$s0-$s7, $t0-$t9, $zero,	Fast locations for data. In MIPS, data must be in registers to perform
32 registers	$a0-$a3, $v0-$v1, $gp,	arithmetic. MIPS register $zero always equals 0. Register $at is
	$fp, $sp, $ra, $at	reserved for the assembler to handle large constants.
	Memory[0],	Accessed only by data transfer instructions. MIPS uses byte addresses, so
2^{30} memory	Memory[4], ...,	sequential words differ by 4. Memory holds data structures, such as arrays,
words	Memory[4294967292]	and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Review

- MIPS defines instructions to be same size as data (one word) so that they can use the same memory (can use `lw` and `sw`).

◦ Machine Language Instruction: 32 bits representing a single instruction

R	`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`
I	`opcode`	`rs`	`rt`	`immediate`		

Computer actually stores programs as a series of these.

Outline

- **Review branch instruction encoding**
- **Jump instructions**
- **Disassembly**
- **Pointers to structures**

Branches: PC-Relative Addressing

- **Branch Calculation:**
  - If we don't take the branch:

  PC = PC + 4

  - If we do take the branch:

  PC = (PC + 4) + (`immediate * 4`)

  - Observations

    1. `Immediate` field specifies the number of words to jump, which is simply the number of instructions to jump.
    2. `Immediate` field can be positive or negative.
    3. Due to hardware, add `immediate` to (PC+4), not to PC; will be clearer why later in course

J-Format Instructions (1/2)

- **For branches, we assumed that we won't want to branch too far, so we can specify *change* in PC.**
- **For jumps (`j` and `jal`), we may jump to *anywhere* in memory.**
- **Ideally, we could specify a 32-bit memory address to jump to.**
- **Unfortunately, we can't fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.**
- **Define 'fields' of the following number of bits each:**

6 bits	**26 bits**

- **As usual, each field has a name:**

opcode	**target address**

- **Key Concepts**

  1. **Keep opcode field same as R-format and I-format for consistency.**
  2. **Combine all other fields to make room for target address.**

J-Format Instructions (2/2)

- **We can specify 28 bits of the 32-bit address, by using WORD address.**
- **Where do we get the other 4 bits?**
  - By definition, take the 4 highest order bits from the PC.
  - Technically, this means that we cannot jump to *anywhere* in memory, but it's adequate 99.9999% of the time, since programs aren't that long.
  - If we absolutely need to specify a 32-bit address, we can always put it in a register and use the `jr` instruction.
- **Summary:**
  - New PC = PC[31..28] || target address (26bits)|| 00
  - Note: II means concatenation
    4 bits || 26 bits || 2 bits = 32-bit address
- **Understand where each part came from!**

Outline

- Review branch instruction encoding
- Jump instructions
- **Disassembly**
- Pointers to structures

Decoding Machine Language

- **How do we convert 1s and 0s to C code?**

  Machine language => MAL => C

- **For each 32 bits:**
  - Look at `opcode`: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format.
  - Use instruction type to determine which fields exist.
  - Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.
  - Logically convert this MIPS code into valid C code. Always possible? Unique?

Decoding Example (1/6)

- **Here are six machine language instructions in hex:**

  00001025

  0005402A

  11000003

  00441020

  20A5FFFF

  08100001

- **Let the first instruction be at address $4{,}194{,}304_{10}$ (0x00400000).**
- **Next step: convert to binary**

Decoding Example (2/6)

- The six machine language instructions in binary:

```
00000000000000000001000000100101

00000000000001010100000000101010

00010001000000000000000000000011

00000000010001000001000000100000

00100000101001011111111111111111

00000100000100000000000000000001
```

- Next step: separation of fields

Decoding Example (3/6)

- **Fields separated based on opcode:**

0	0	0	2	0	37
0	0	5	8	0	42
4	8	0	+3		
0	2	4	2	0	32
8	5	5	-1		
2	00000100000000000000000001				

Next step: translate to MIPS instructions

Decoding Example (4/6)

- **MIPS Assembly (Part 1):**

```
0x00400000   or     $2,$0,$0
0x00400004   slt    $8,$0,$5
0x00400008   beq    $8,$0,3
0x0040000c   add    $2,$2,$4
0x00400010   addi   $5,$5,-1
0x00400014   j      0x100001
```

Better solution: translate to more meaningful instructions (fix the branch/jump and add labels)

Decoding Example (5/6)

- **MIPS Assembly (Part 2):**

```
          or    $v0,$0,$0
Loop:     slt   $t0,$0,$a1
          beq   $t0,$0,Fin
          add   $v0,$v0,$a0
          addi  $a1,$a1,-1
          j     Loop
Fin:
```

Next step: translate to C code (be creative!)

Decoding Example (6/6)

- **C code:**

Mapping:	$v0: product
	$a0: mcand
	$a1: mplier

```
product = 0;
  while ( mplier > 0) {
      product += mcand;
      mplier -= 1;
  }
```

Outline

- **Loading/Storing Bytes**
- Signed vs. Unsigned MIPS Instructions
- Pseudo-instructions
- Multiply/Divide
- Pointers and assembly language

Loading, Storing bytes

- **In addition to word data transfers (`lw`, `sw`), MIPS has byte data transfers:**
- **load byte: `lb`**
- **store byte: sb**
- **same format as `lw`, `sw`**
- **What do with other 24 bits in the 32 bit register?**
  - `lb`: sign extends to fill upper 24 bits
- **Suppose byte at 100 has value 0x0F, byte at 200 has value 0xFF**

```
lb $s0, 100($zero) # $s0 = ??
lb $s1, 200($zero) # $s1 = ??
```

- **Multiple choice: $s0? $s1?**

  a) 15; b) 255; c) -1; d) -255; e) -15

Loading bytes

- **Normally with characters don't want to sign extend**
- **MIPS instruction that doesn't sign extend when loading bytes:**

  load byte unsigned: lbu

Outline

- Loading/Storing Bytes
- **Signed vs. Unsigned MIPS Instructions**
- Pseudo-instructions
- Multiply/Divide
- Pointers and assembly language

Overflow in Arithmetic (1/2)

- **Reminder: Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers.**

- **Example (4-bit unsigned numbers):**

```
+15          1111
 +3          0011
------       ----
+18         10010
```

  - But we don’t have room for 5-bit solution, so the solution would be 0010, which is +2, and wrong.

Overflow in Arithmetic (2/2)

- **MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:**
  - add (`add`), add immediate (`addi`) and subtract (`sub`) cause overflow to be detected
  - add unsigned (`addu`), add immediate unsigned (`addiu`) and subtract unsigned (`subu`) do not cause overflow detection
- **Compiler selects appropriate arithmetic**
  - MIPS C compilers produce `addu`, `addiu`, `subu`

Unsigned Inequalities

- **Just as unsigned arithmetic instructions:**

```
addu, subu, addiu
```

(really "don't overflow" instructions)

- **There are unsigned inequality instructions:**

```
sltu, sltiu
```

but really do mean unsigned compare!

0x80000000 < 0x7fffffff signed (slt, slti)

0x80000000 > 0x7fffffff unsigned (sltu,sltiu)

Outline

- Loading/Storing Bytes
- Signed vs. Unsigned MIPS Instructions
- **Pseudo-instructions**
- Multiply/Divide
- Pointers and assembly language

True Assembly Language

- **Pseudo-instruction: A MIPS instruction that doesn t turn directly into a machine language instruction.**

- **What happens with pseudoinstructions?**
  - They’re broken up by the assembler into several ‘real’ MIPS instructions.
  - But what is a ‘real’ MIPS instruction?

Example Pseudoinstructions

- **Register Move**

  ```
  move  reg2,reg1
  ```

  Expands to:

  ```
  add   reg2,$zero,reg1
  ```

- **Load Immediate**

  ```
  li reg,value
  ```

  If value fits in 16 bits:

  ```
  ori   reg,$zero,value
  else:
  lui   reg,upper 16 bits of value
  ori   reg,$zero,lower 16 bits
  ```

True Assembly Language

- **MAL (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this *includes* pseudoinstructions**

- **TAL (True Assembly Language): the set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)**

- **A program must be converted from MAL into TAL before it can be translated into 1s and 0s.**

Where are MIPS processors?

Digital Entertainment:

- Set-top Boxes, Personal Video Recorders, Game Consoles, Digital Television

Mobile Computing:

- Palm & Pocket PCs, Handheld PCs, Cars

Office Automation:

- **Printers, Copiers, Network Computers, Scanners**

Consumer Electronics:

- **Digital Cameras, GPS Surveying, Smart Phones, Smart Cards, Robotic Toys**

Communications/Networking:

- **Routers, Network Cards, Internet Servers**

Outline

- Loading/Storing Bytes
- Signed vs. Unsigned MIPS Instructions
- Pseudo-instructions
- **Multiply/Divide**
- Pointers and assembly language

Multiplication (1/3)

- **Paper and pencil example (unsigned):**

```
Multiplicand       1000      8
Multiplier        x1001      9
                   1000
                  0000
                 0000
               +1000
               01001000
```

m bits x n bits = m + n bit product

Multiplication (2/3)

- **In MIPS, we multiply registers, so:**
  - 32-bit value x 32-bit value = 64-bit value
- **Syntax of Multiplication:**
  - `mult` register1, register2
  - Multiplies 32-bit values in specified registers and puts 64-bit product in special result registers:
    - puts upper half of product in hi
    - puts lower half of product in lo
  - hi and lo are 2 registers separate from the 32 general purpose registers

Multiplication (3/3)

- **Example:**
  - in C: `a = b * c;`
  - in MIPS:

    let b be $s2; let c be $s3; and let a be $s0 and $s1 (since it may be up to 64 bits)

```
mult $s2,$s3    # b * c
mfhi $s0        # upper half of product into $s0
mflo $s1        # lower half of product into $s1
```

- **Note: Often, we only care about the lower half of the product.**

Division (1/3)

- **Paper and pencil example (unsigned):**

```
                         1001
Quotient Divisor 1000|1001010 Dividend
                     -1000
                         1010
                        -1000
                           10
Remainder    (or Modulo result)
```

- **Dividend = Quotient x Divisor + Remainder**

Division (2/3)

- **Syntax of Division:**
  - `div` register1, register2
  - Divides 32-bit values in register 1 by 32-bit value in register 2:

    puts remainder of division in hi

    puts quotient of division in lo
- **Notice that this can be used to implement both the C division operator (/) and the C modulo operator (%)**

Division (3/3)

- **Example:**
  - in C: `a = c / d;`
    `b = c % d;`
  - in MIPS:

  let a be $s0; let b be $s1; let c be $s2; and let d be $s3

```
div  $s2,$s3    # lo=c/d, hi=c%d
mflo $s0        # get quotient
mfhi $s1        # get remainder
```

More Overflow Instructions

- **In addition, MIPS has versions of these two arithmetic instructions for unsigned operands:**

```
multu

divu
```

Outline

- Loading/Storing Bytes
- Signed vs. Unsigned MIPS Instructions
- Pseudo-instructions
- Multiply/Divide
- **Pointers and assembly language**

Address vs. Value

- Fundamental concept of Comp. Sci.
- Even in Spreadsheets: select cell A1 for use in cell B1

	A	B
1	100	100
2		

Do you want to put the address of cell A1 in formula (=A1) or A1’s value (100)?

Difference? When change A1,
cell using address changes,
but not cell with old value

Assembly Code to Implement Pointers

- **deferencing Þ data transfer in asm.**
  - ... = ... *p ...; Þ load
    (get value from location pointed to by p)
    load word (lw) if int pointer,
    load byte unsigned (lbu) if char pointer
  - *p = ...; Þ store
    (put value into location pointed to by p)

Assembly Code to Implement Pointers

c is int, has value 100, in memory at address 0x10000000, p in $a0, x in $s0

```
 p = &c;  /* p gets 0x10000000 */
 x = *p;  /* x gets 100 */
*p = 200; /* c gets 200 */
```

```
# p = &c;  /* p gets 0x10000000 */
 lui $a0,0x1000 # p = 0x10000000

# x = *p;  /* x gets 100 */
 lw  $s0, 0($a0) # dereferencing p

# *p = 200; /* c gets 200 */
 addi $t0,$0,200
 sw   $t0, 0($a0) # dereferencing p
```

Registers and Pointers

- **Registers do not have addresses**
  - Þ registers cannot be pointed to
  - Þ cannot allocate a variable to a register
    if it may have a pointer to it

C vs. Asm with Pointer Arithmetic

```
int strlen(char *s) {
  char *p = s;          /* p points to chars */

 while (*p != ‘\0’)
    p++;                /* points to next char */
  return p - s;         /* end - start */

}
```

```
        mov  $t0,$a0
        bu   $t1,0($t0)         /* derefence p */
        eq   $t1,$zero, Exit

Loop:   addi $t0,$t0,1          /* p++ */
        lbu $t1,0($t0)          /* derefence p */
        bne  $t1,$zero, Loop

Exit:   sub $v0,$t1,$a0
        jr $ra
```

And in Conclusion..

- **MIPS Signed v. Unsigned "overloaded" term**

  Do/Don't sign extend (lb, lbu)

  Don't overflow (addu, addiu, subu, multu, divu)

  Do signed/unsigned compare (slt,slti/sltu,sltiu)

- **Assembler uses $at to turn MAL into TAL**

- **MIPS mult/div instructions use hi, lo registers.**

- **Pointer dereferencing directly supported as load/store.**