ISA (Instruction Set Architecture) Part III

Ergun Simsek

CMSC 411 | Lecture 7 | Fall 2022

1

Last class

- MIPS Instructions
- How to use the MIPS Reference Card (Green Card)

Today

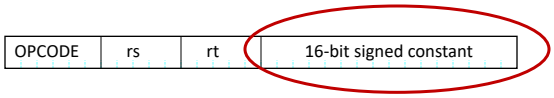
- MIPS Instructions: Branching, Multiplication, Division, Logical
- Accessing Memory
- Addressing Modes

2

MIPS Branch Instructions

MIPS *branch instructions* provide a way of conditionally changing the PC to some nearby location...

I-type:




```
beq rs, rt, label  # Branch if equal

    if (REG[RS] == REG[RT])
     {
       PC = PC + 4 + 4*offset;
     }
```

```
bne rs, rt, label  # Branch if not equal

    if (REG[RS] != REG[RT])
     {
       PC = PC + 4 + 4*offset;
     }
```

Notice on memory references offsets are multiplied by 4, so that branch targets are restricted to word boundaries.

NB: Branch targets are specified relative to the next instruction (which would be fetched by default). The assembler hides the calculation of these offset values from the user, by allowing them to specify a target address (usually a label) and it does the job of computing the offset's value. The size of the constant field (16-bits) limits the range of branches.

3

3

MIPS Jumps

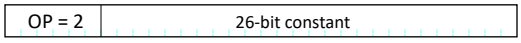
The range of MIPS branch instructions is limited to approximately ± 32K instructions (± 128K bytes) from the branch instruction.
To branch farther: an unconditional jump instruction is used.

Instructions:

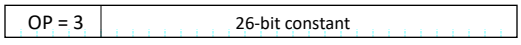
```
j label          # jump to label (PC = { PC[31-28], CONST[25:0]*4) }
                         lower 28 bits are the const * 4
                         upper 4 bits are from the current PC value
                         " { } " here means concatenate them together
jal label        # jump to label and store PC+4 in $31
jr $t0           # jump to address specified by register's contents
jalr $t0, $ra    # jump to address specified by first register's contents
                   and store PC+4 in second register
```

Formats:

- J-type: used for j



- J-type: used for jal



- R-type, used for jr

- R-type, used for jalr

4

4

Multiply and Divide

Slightly more complicated than add/subtract

- multiply: product is twice as long!
  - if A, B are 32-bit long, A * B is how many bits?
- divide: dividing integer A by B gives two results!
  - quotient and remainder

Solution: two new special-purpose registers

- "Hi" and "Lo"

5

Multiply: MULT instruction

`mult rs, rt`

- Meaning: multiply contents of registers $rs and $rt, and store the (64-bit result) in the pair of special registers **`{hi, lo}`**

  `hi:lo = $rs * $rt`

- upper 32 bits go into **`hi`**, lower 32 bits go into **`lo`**

To access result, use two new instructions

- **`mfhi:`** move from hi, i.e., move the 32-bit half result from hi to $rd

  `mfhi rd`

- **`mflo:`** move from lo, i.e., move the 32-bit half result from **`lo`** to $rd

  `mflo rd`

```
mult $a0, $a1
mfhi $a2        # 32 most significant bits of multiplication to $a2
mflo $v0        # 32 least significant bits of multiplication to $v0
```


6

Divide: DIV instruction

```
div rs, rt
```

- Meaning: divide contents of register $rs by $rt, and store the quotient in `lo`, and remainder in `hi`

```
lo = $rs / $rt
hi = $rs % $rt
```

To access result, use **`mfhi`** and **`mflo`**

NOTE: There are also unsigned versions

- **`multu`**
- **`divu`**

Comparison: `slt, slti`

slt = set-if-less-than

- **`slt rd, rs, rt`**

  $rd = ($rs < $rt) // "1" if true and "0" if false

slti = set-if-less-than-immediate

- **`slti rt, rs, imm`**

  $rt = ($rs < sign-ext(imm))

also other flavors

- **`sltu`**
- **`sltiu`**

Logical Instructions

Boolean operations: bitwise on all 32 bits

- operations: AND, OR, NOR, XOR
- instructions:
  - `and, andi`
  - `or, ori`
  - `nor` // Note: There is no `nori`
  - `xor, xori`

X	Y	AND(X,Y)	OR(X,Y)	NAND(X,Y)	NOR(X,Y)	XOR(X,Y)
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Examples:

- `and $1, $2, $3`
  $1 = $2 & $3
- `xori $1, $2, 0xFF12`
  $1 = $2 ^ 0x0000FF12
- See all in textbook!

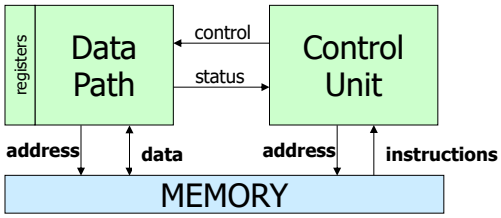
CMSC 411 – Lecture 7 9

9

Accessing Memory

MIPS is a "load-store" architecture

- all operands for ALU instructions are in registers or immediate
- cannot directly add values residing in memory
  - must first bring values into registers from memory (called LOAD)
  - must store result of computation back into memory (called STORE)

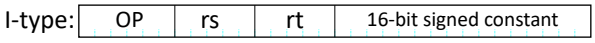


CMSC 411 – Lecture 7 10

10

MIPS Load Instruction

Load instruction is I-type



`lw rt, imm(rs)`

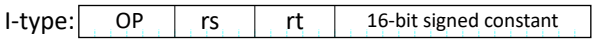
Meaning: Reg[rt] = Mem[Reg[rs] + sign-ext(imm)]

Abbreviation: `lw rt,imm` for `lw rt, imm($0)`

- Does the following:
  - takes the value stored in register $rs
  - adds to it the immediate value (signed)
  - this is the address where memory is looked up
  - value found at this address in memory is brought in and stored in register $rt

MIPS Store Instruction

Store instruction is also I-type



`sw rt, imm(rs)`

Meaning: Mem[Reg[rs] + sign-ext(imm)] = Reg[rt]

Abbreviation: `sw rt,imm` for `sw rt, imm($0)`

- Does the following:
  - takes the value stored in register $rs
  - adds to it the immediate value (signed)
  - this is the address where memory is accessed
  - reads the value from register $rt and writes it into the memory at the address computed

MIPS Memory Addresses

`lw` and **`sw`** read whole 32-bit words

- so, addresses computed must be multiples of 4
  - Reg[rs] + sign-ext(imm) must end in "00" in binary
- otherwise: runtime exception

There are also byte-sized flavors of these instructions

- **`lb`** (load byte)
  - Loads the byte from memory into the low order eight bits of the register. These are bits 0-7 of the register.
  - Then it copies bit 7 to bits 8-31 of the register (all bits to the left of bit 7).
- **`sb`** (store byte)
- lb/sb addresses do not have to be multiples of 4

Example

- Memory at 0x10000007 contains the byte 0xA4
- Register $8 contains 0x10000000
- What is put in register $10 when the following instruction is executed:

  lb $10,7($8)

  $10 = 0xFFFFFFA4
  Bit 7 of 0xA4 is 1, so lb extends that bit to all high order three bytes of $10.

- What is put in register $12 when the following instruction is executed:

  lbu $12,7($8)

  $12 = 0x000000A4
  lbu zero-extends the byte from memory.

Storage Conventions

Data stored in memory

- addresses in memory are assigned at compile time
- data values must be "loaded" into registers first
- operations done on registers
- result stored in memory

```
int x, y;
y = x + 37;
```

Compilation approach: LOAD, COMPUTE, STORE

translates to:

```
lw    $t0, 0x1008($0)
addi  $t0, $t0, 37
sw    $t0, 0x100C($0)
```

choice of $t0 is arbitrary

Example

- assume compiler has assigned these memory addresses

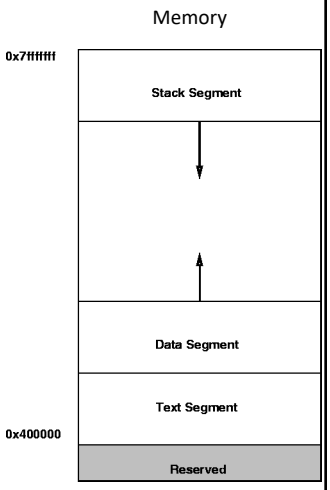
1000:	n
1004:	r
1008:	x
100C:	y
1010:	

CMSC 411 – Lecture 7 15

15

Some Notes About PC

- Both data and text (instructions) are stored in memory.
- However, these two quantities are stored in different areas of memory, and accessed by different data pathways.
- The MIPS computer can address 4 Gbyte of memory, from address 0x0000 0000 to 0xffff ffff.
- User memory is limited to locations below 0x7fff ffff.
- Text (program) storage always starts at 0x0040 0000.
- Data storage generally* starts at 0x1001 0000.
- Data and instructions enter the CPU via different pathways.
  - Data must enter via the register block.
  - Instructions proceed directly to the instruction decoder in the CPU



* Different implementations use slightly different address

CMSC 411 – Lecture 7 16

16

Addressing Modes

Today's modern CPUs can have very complex addressing modes, but MIPS is not one of them.

In MIPS, there are three modes

Absolute (Direct): `lw $8, 0x1000($0)`
- Value = Mem[constant]
- Use: accessing static data

Register-Indirect: `lw $8, 0($9)`
- Value = Mem[Reg[x]]
- Use: pointer accesses

Displacement: `lw $8, 16($9)`
- Value = Mem[Reg[x] + constant]
- Use: access to local variables

Do you want to see a complex one? ➡ Scaled:
- Value = Mem[Reg[x] + c + d*Reg[y]]
- Use: array accesses (base+index)

Absolute (Direct) Addressing

What we want:
- Contents of a specific memory location, i.e. at a given address

Example:

"C"
```
int x = 10;

main() {
    x = x + 1;
}
```

"MIPS Assembly"
```
.data
x: .word 10

.text
main:
        lw   $2,x($0)
        addi $2,$2,1
        sw   $2,x($0)
```

Allocates space for a single integer (4-bytes) and initializes its value to 10

'x' here means address of x

Warning
- Sometimes generates a two-instruction sequence
  - If the address for `x` chosen by the programmer/compiler/assembler is too large to fit within the 16-bit immediate field

```
lui  $1,xhighbits
lw   $2,xlowbits($1)
```

e.g., if x is at address 0x12345678 ➡

```
lui  $1,0x1234
lw   $2,0x5678($1)
```

Absolute (Direct) Addressing: More detail

"C"

```
int x = 10;

main() {
    x = x + 1;
}
```

"MIPS Assembly"

```
.data
x: .word 10

.text
main:
       lw   $2,x($0)
       addi $2,$2,1
       sw   $2,x($0)
```

"After Compilation"

```
.data 0x0100
x: .word 10

.text
main:
       lw   $2,0x100($0)
       addi $2,$2,1
       sw   $2,0x100($0)
```

- Assembler replaces "**x**" by its address
  - e.g., here the data part of the code (**.data**) starts at 0x100
  - **x** is the first variable, so starts at 0x100
    - this mode works in MIPS only if the address fits in 16 bits

What does this code do?

"C"

```
int x = 10;

main() {
    int *y = &x;
    *y = 2;
}
```

x is the address of the value, which is 10

&x is the address of the pointer that contains the address of the value

***y** is the value

Indirect Addressing

What we want:

- The contents at a memory address held in a register

Examples

"C"

```
int x = 10;

main() {
    int *y = &x;
    *y = 2;
}
```

"MIPS Assembly"

```
.data
x: .word   10

.text
main:
        la   $2,x
        addi $3,$0,2
        sw   $3,0($2)
```

la: load address
"la" is not a real instruction,
It's a convenient *pseudoinstruction* that constructs a constant via either a 1 instruction or 2 instruction sequence

```
ori   $2,$0,x

lui   $2,xhighbits
ori   $2,$2,xlowbits
```

$2 performs the role of y, i.e., it is a pointer to x

Warning

- You must make sure that the register contains a valid address (double, word, or short aligned as required)

Note on `la` pseudoinstruction

`la` is a pseudoinstruction: `la $r, x`

- stands for "load the address of" variable `x` into register `r`
- not an actual MIPS instruction
- but broken down by the assembler into actual instructions
  - if address of x is small (fits within 16 bits), then a single `ori`
    - one could also use a single `addiu`
  - if address of x is larger, use the `lui + ori` combo

"MIPS Assembly"

```
.data
x: .word   10

.text
.global main
main:
        la   $2,x
        addi $3,$0,2
        sw   $3,0($2)
```

if `0x0100`

```
ori   $2,$0,0x100
```

if `0x80000010`

```
lui   $2,0x8000
ori   $2,$2,0x0010
```

Displacement Addressing

What we want:

- contents of a memory location at an offset relative to a register
  - the address that is the sum of a constant and a register value

Example:

a[4]	a[3]	a[2]	a[1]	a[0]
19-16	15-12	11-8	7-4	3-0

"C"

```
int a[5];

main() {
    int i = 3;
    a[i] = 2;
}
```

"MIPS Assembly"

```
.data
a: .space   20

.text
main:
    addi $2,$0,3   // i in $2
    addi $3,$0,2
    sll  $1,$2,2   // i*4 in $1
    sw   $3,a($1)
```

Allocates space for a 5 uninitialized integers (20-bytes)

Remember:

- Must multiply (shift) the "index" to be properly aligned

Displacement Addressing: 2nd example

What we want:

- The contents of a memory location at an offset relative to a register

Example:

"C"

```
struct p {
   int x, y; }

main() {
    p.x = 13;
    p.y = 12;
}
```

"MIPS Assembly"

```
.data
p: .space 8

.text
main:
    la   $1,p
    addi $2,$0,13
    sw   $2,0($1)
    addi $2,$0,12
    sw   $2,4($1)
```

Allocates space for 2 uninitialized integers (8-bytes)

Note:

- offsets to the various fields within a structure are constants known to the assembler/compiler

Practice Problem: Store offsets

The variable A is stored at **memory address 24**. We want to change its value to 512. Which of the following combinations of **R5**, **R6**, and the constant **X** will accomplish this with the instruction **sw R6, X(R5)**?

1. R6=512, X=0, R5=24
2. R6=512, X=24, R5=0
3. R6=24, X=512, R5=0
4. R6=24, X=0, R5=512