ECE 154A Introduction to Computer Architecture
Fall 2012

Dmitri Strukov

Lecture 5: Control Flow Instructions / Multiplication and Division
Agenda

• Review of last lecture (Data Transfer Instructions)
• Control Flow Instructions
• Multiply and divide instructions
Last Lecture
CPU overview
Memory Operands

• Main memory used for composite data
  – Arrays, structures, dynamic data
• To apply arithmetic operations
  – Load values from memory into registers
  – Store result from register to memory
• Memory is byte addressed
  – Each address identifies an 8-bit byte
• Words are aligned in memory
  – Address must be a multiple of 4
• MIPS is Big Endian
  – Most-significant byte at least address of a word
  – c.f. Little Endian: least-significant byte at least address
Data Transfer: Memory to Register

- MIPS load Instruction Syntax
  \[ lw \text{ register#}, \text{offset(register#)} \]
  (1)    (2)    (3)    (4)

Where
1) operation name
2) register that will receive value
3) numerical offset in bytes
4) register containing pointer to memory

\textit{lw} – meaning Load Word
32 bits or one word are loaded at a time
Data Transfer: Register to Memory

• MIPS store Instruction Syntax

```plaintext
sw register#, offset(register#)
```

Where

1) operation name
2) register that will be written in memory
3) numerical offset in bytes
4) register containing pointer to memory

sw – meaning Store Word
32 bits or one word are stored at a time
Memory Operand Example 1

• C code:
  
g = h + A[8];
  – g in $s1, h in $s2, base address of A in $s3

• Compiled MIPS code:
  – Index 8 requires offset of 32
    • 4 bytes per word

  lw $t0, 32($s3)    # load word
  add $s1, $s2, $t0
Memory Operand Example 2

• C code:
  
  \[ A[12] = h + A[8]; \]
  
  – h in $s2, base address of A in $s3

• Compiled MIPS code:
  
  – Index 8 requires offset of 32

  \[
  \text{lw} \quad $t0, \quad 32($s3) \quad \# \text{load word}
  \]

  \[
  \text{add} \quad $t0, \quad $s2, \quad $t0 \quad \# \text{add}
  \]

  \[
  \text{sw} \quad $t0, \quad 48($s3) \quad \# \text{store word}
  \]
Registers vs. Memory

• Registers are faster to access than memory

• Operating on memory data requires loads and stores
  – More instructions to be executed

• Compiler must use registers for variables as much as possible
  – Only spill to memory for less frequently used variables
  – Register optimization is important!
Byte/Halfword Operations

• MIPS byte/halfword load/store
  – String processing is a common case
    \[ \text{lb} \ rt, \ \text{offset} (rs) \quad \text{lh} \ rt, \ \text{offset} (rs) \]
  – Sign extend to 32 bits in rt
    \[ \text{lbu} \ rt, \ \text{offset} (rs) \quad \text{lhu} \ rt, \ \text{offset} (rs) \]
  – Zero extend to 32 bits in rt
    \[ \text{sb} \ rt, \ \text{offset} (rs) \quad \text{sh} \ rt, \ \text{offset} (rs) \]
  – Store just rightmost byte/halfword

Why do we need them?
→ characters and multimedia data are expressed by less than 32 bits; having dedicated 8 and 16 bits load and store instructions results in faster operation
Control Flow Instructions
Conditional Operations

• Branch to a labeled instruction if a condition is true
  – Otherwise, continue sequentially
• beq rs, rt, L1
  – if (rs == rt) branch to instruction labeled L1;
• bne rs, rt, L1
  – if (rs != rt) branch to instruction labeled L1;
• j L1
  – unconditional jump to instruction labeled L1
Compiling If Statements

• C code:
  
  ```
  if (i == j) f = g + h;
  else f = g - h;
  ```

  − f, g, ... in $s0, $s1, ...

• Compiled MIPS code:
  
  ```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  ```

  ```
  Else:
  sub $s0, $s1, $s2
  Exit: ...
  ```

Assembler calculates addresses
Compiling Loop Statements

• C code:

```c
while (save[i] == k) i += 1;
```

- i in $s3, k in $s5, address of save in $s6

• Compiled MIPS code:

```mips
Loop:    sll $t1, $s3, 2      #t1 = i * 4
        add $t1, $t1, $s6
        lw $t0, 0($t1)
        bne $t0, $s5, Exit
        addi $s3, $s3, 1
        j Loop
Exit:    ...
```
More Conditional Operations

• Set result to 1 if a condition is true
  – Otherwise, set to 0
• \texttt{slt}\ rd, rs, rt
  – if (rs < rt) rd = 1; else rd = 0;
• \texttt{slti}\ rt, rs, constant
  – if (rs < constant) rt = 1; else rt = 0;
• Use in combination with beq, bne
  \texttt{slt}\ \$t\ 0, \$s1, \$s2 # if ($s1 < $s2)
  \texttt{bne}\ \$t\ 0, \$zero, L # branch to L
Branch Instruction Design

• Why not bl t, bge, etc?
• Hardware for <, ≥, ... slower than =, ≠
  – Combining with branch involves more work per instruction, requiring a slower clock
  – All instructions penalized!
• beq and bne are the common case
• This is a good design compromise
More Examples: For Loop

C code:  for(i = 1; i <= 10; i++)  A[i] = A[i] + 1;

Okay assembly code:  assume i: $s1, base of A: $s2

```assembly
addi $s1, $0, 1  # i = 1

LOOP:
slti $t0, $s1, 11  # LOOP: if(i < 11) $t0 = 1 else $t0 = 0
beq $t0, $0, END  # if ($t0==0) goto END; else {
  sll $t0, $s1, 2  # $t0 = i * 4;
  add $t0, $s2, $t0  # $t0 = $t0 + addr A;
  lw $t1, 0($t0)   # $t1 = A[i];
  addi $t1, $t1, 1 # $t1 = $t1 + 1;
  sw $t1, 0($t0)   # A[i] = $t1;
  addi $s1, $s1, 1 # i = i + 1;
  j LOOP           # goto LOOP }

END:              # END:
```

2 control flow instructions + 7 other instructions in the loop
More Examples: For Loop

C code:  
```c
for(i = 1; i <= 10; i++)  A[i] = A[i] + 1;
```

Better assembly code: assume $i:$s1, base of $A:$s2

```assembly
addi $s1, $0, 1       # i = 1;
addi $t0, $s2, 4      # $t0 = addr A + 4; (* pointer to A[1] *)
addi $t2, $0, 11      # $t2 = 11;

LOOP: lw  $t1, 0($t0)       # do { $t1 = A[i];
addi $t1, $t1, 1       #    $t1 = $t1 + 1;
sw  $t1, 0($t0)         #    A[i] = $t1;
addi $s1, $s1, 1       #    i = i + 1;
addi $t0, $t0, 4       #    $t0 = $t0 + 4; }
be  $s1, $t2, LOOP      # while (i != 11);
```

1 control flow instructions + 5 other instructions in the loop
More Examples: For loop

C code:  for(i = 1; i <= 10; i++)  A[i] = A[i] + 1;

Even better assembly code:  assume i: $s1, base of A: $s2

```
addi $t0, $s2, 4             # $t0 = addr A + 4; (* pointer to a[1] *)
addi $t2, $t0, 40            # $t2 = $t0 + 40; (* pointer to a[11] *)

LOOP:                        # do { $t1 = A[i];
    lw   $t1, 0($t0)          #    $t1 = $t1 + 1;
    addi $t1, $t1, 1          #    A[i] = $t1;
    sw   $t1, 0($t0)          #    $t0 = $t0 + 4; }
    addi $t0, $t0, 4          # $t0 = $t0 + 4; }
    bne  $t0, $t2, LOOP       # while ($t2 != $t0);
    addi $s1, $0, 11          # i = 11
```

(note that in this case the variable i is not used at all. The last line is just to make C code functionally equivalent to assembly code, since in C variable i will be equal to 11 after the completion of the loop)

1 control flow instructions + 4 other instructions in the loop
Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt $t0, $s0, $s1` # signed
    - $-1 < +1 \Rightarrow t0 = 1$
  - `sltu $t0, $s0, $s1` # unsigned
    - $+4,294,967,295 > +1 \Rightarrow t0 = 0$
Review

• Instructions so far:
  add, addi, sub
  lw, sw,
  lb, lbu, lh, lhu
  beq, bne, j
  slt, slti, sltu, sltui

• Registers so far
  C variables: $s0 - $s7
  Temporary variables: $t0 - $t9
  Zero: $zero