Lecture 6: Data Transfer & Control Instructions

CSE 30: Computer Organization and Systems Programming
Winter 2010

Rajesh Gupta/Ryan Kastner
Dept. of Computer Science and Engineering
University of California, San Diego
Today

- Data Transfer Instructions and Alignment
  - Load, Store
- Control Instructions
Assembly Operands: Memory

- C variables map onto registers; what about large data structures like arrays?
- 1 of 5 components of a computer: memory contains such data structures
- But MIPS arithmetic instructions only operate on registers, never directly on memory.
- **Data transfer instructions** transfer data between registers and memory:
  - Memory to register
  - Register to memory
Anatomy: 5 components of any Computer

Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.

These are “data transfer” instructions…
Data Transfer: Memory to Register

- To transfer a word of data, we need to specify two things:
  - Register: specify this by number (0 - 31) or symbolic name ($s0,…, $t0, …)
  - Memory address: more difficult
    - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
    - Other times, we want to be able to offset from this pointer.

Remember: Load FROM memory
Data Transfer: Memory to Register

- To specify a memory address to copy from, specify two things:
  - A register which contains a pointer to memory
  - A numerical offset (in bytes)
- The desired memory address is the sum of these two values.
- Example: \(8(\$t0)\)
  - specifies the memory address pointed to by the value in \$t0, plus 8 bytes
Data Transfer: Memory to Register

Load Instruction Syntax:

```
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

MIPS Instruction Name:

```
1w (meaning Load Word, so 32 bits or one word are loaded at a time)
```
Data Flow

Example: \texttt{lw \$t0,12(\$s0)}

This instruction will take the pointer in \$s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register \$t0

Notes:

\$s0 is called the \texttt{base register}

12 is called the \texttt{offset}

offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure
Data Transfer: Register to Memory

- Also want to store value from a register into memory
- Store instruction syntax is identical to Load instruction syntax
- MIPS Instruction Name: `sw` (meaning Store Word, so 32 bits or one word are loaded at a time)
- Example: `sw $t0,12($s0)`
  This instruction will take the pointer in `$s0`, add 12 bytes to it, and then store the value from register `$t0` into the memory address pointed to by the calculated sum

Remember: Store INTO Memory
Pointers vs. Values

- **Key Concept:** A register can hold any 32-bit value. That value can be a (signed) `int`, an unsigned `int`, a pointer (memory address), and so on.

- If you write `add $t2, $t1, $t0` then `$t0` and `$t1` better contain values.

- If you write `lw $t2, 0($t0)` then `$t0` better contain a pointer.

- Don’t mix these up!
Addressing: Byte vs. word

- Every word in memory has an address, similar to an index in an array.
- Early computers numbered words like C numbers elements of an array:
  - Memory[0], Memory[1], Memory[2], ...
  - Called the “address” of a word
- Computers needed to access 8-bit (byte) as well as words (4 bytes/word).
- Today machines address memory as bytes, hence 32-bit (4 byte) word addresses differ by 4:
  - Memory[0], Memory[4], Memory[8], ...
Compilation with Memory

- What offset in `lw` to select `A[8]` in C?
- `4x8=32` to select `A[8]`: byte v. word

Compile by hand using registers:

\[ g = h + A[8]; \]

\[ g: \$s1, h: \$s2, \$s3: \text{base address of A} \]

- 1st transfer from memory to register:
  \[ lw \ \$t0,32(\$s3) \# \$t0 \text{ gets } A[8] \]
  \[ \text{Add 32 to } \$s3 \text{ to select } A[8], \text{ put into } \$t0 \]

- Next add it to `h` and place in `g`
  \[ \text{add } \$s1,\$s2,\$t0 \# \$s1 = h+A[8] \]
Notes about Memory

- Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
  
  - Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
  
  - So remember that for both \texttt{lw} and \texttt{sw}, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)
More Notes about Memory: Alignment

- MIPS requires that all words start at byte addresses that are multiples of 4 bytes.

```
0 | 1 | 2 | 3
---|---|---|---
Aligned
|   |   |   |   
Not Aligned
|   |   |   |   
```

- Last hex digit of address is:
  - 0, 4, 8, or \( C_{hex} \)
  - 1, 5, 9, or \( D_{hex} \)
  - 2, 6, A, or \( E_{hex} \)
  - 3, 7, B, or \( F_{hex} \)

- Called **Alignment**: objects must fall on address that is multiple of their size.
Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variables in registers
  - Less common in memory: spilling
- Why not keep all variables in memory?
  - Smaller is faster:
    - registers are faster than memory
  - Registers more versatile:
    - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - MIPS data transfer only read or write 1 operand per instruction, and no operation
Takeaway So far

- Memory is byte-addressable, but `lw` and `sw` access one word at a time.
- A pointer (used by `lw` and `sw`) is just a memory address, so we can add to it or subtract from it (using offset).
Arithmetic + Data Transfer

- Instructions so far:
  - add, addi,
  - sub, subi,
  - mult, div,
  - lw, sw

- Registers so far:
  - C Variables: $s0 - $s7
  - Temporary Variables: $t0 - $t9
  - Zero: $zero
Control Instructions
C Decisions: if Statements

- 2 kinds of if statements in C
  - if (condition) clause
  - if (condition) clause1 else clause2

- Rearrange 2nd if into following:
  
  ```c
  if (condition) goto L1;
  clause2;
  goto L2;
  L1: clause1;
  L2:
  ```

- Not as elegant as if-else, but same meaning
MIPS Decision Instructions

- **Decision instruction in MIPS:**
  - `beq register1, register2, L1`
  - `beq` is “Branch if (registers are) equal”
    Same meaning as (using C):
    ```c
    if (register1==/register2) goto L1
    ```

- **Complementary MIPS decision instruction**
  - `bne register1, register2, L1`
  - `bne` is “Branch if (registers are) not equal”
    Same meaning as (using C):
    ```c
    if (register1!=register2) goto L1
    ```

- **Called **conditional branches**
MIPS Goto Instruction

- In addition to conditional branches, MIPS has an unconditional branch:
  
  `j label`

- Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition

- Same meaning as (using C):
  
  `goto label`

- Technically, it’s the same as:
  
  `beq $0,$0,label`

  since it always satisfies the condition.
Compiling C if into MIPS

- Compile by hand
  
  ```
  if (i == j) f=g+h;
  else f=g-h;
  ```

- Use this mapping:
  
  ```
  f: $s0, g: $s1, h: $s2, i: $s3, j: $s4
  ```
Compiling C `if` into MIPS

Compile by hand

```c
if (i == j) f=g+h;
else f=g-h;
```

f: $s0, g: $s1, h: $s2, i: $s3, j: $s4

Final compiled MIPS code:

```
beq $s3,$s4,True  # branch i==j
sub $s0,$s1,$s2    # f=g-h(false)
j Fin            # goto Fin
True: add $s0,$s1,$s2  # f=g+h (true)
Fin:
```

Note: Compiler automatically creates labels to handle decisions (branches) appropriately. Generally not found in HLL code.
Loops in C/Assembly

- Simple loop in C; A[] is an array of ints
  
  ```c
  do{
    g = g + A[i];
    i = i + j;
  }while (i != h);
  ```

- Rewrite this as:
  
  ```c
  Loop: g = g + A[i];
  i = i + j;
  if (i != h) goto Loop;
  ```

- Use this mapping:
  
  g: $s1, h: $s2, i: $s3, j: $s4, base of A:$s5
Loops in C/Assembly

- Final compiled MIPS code:

```
Loop: sll $t1, $s3, 2  #$t1= 4*i
add $t1, $t1, $s5  #$t1=addr A
lw $t1, 0($t1)  #$t1=A[i]
add $s1, $s1, $t1  #g=g+A[i]
add $s3, $s3, $s4  #i=i+j
bne $s3, $s2, Loop  # goto Loop
    # if i!=h
```

- g: $s1, h: $s2, i: $s3, j: $s4, base of A:$s5
Loops in C/Assembly

- There are three types of loops in C:
  - `while`
  - `do... while`
  - `for`

- Each can be rewritten as either of the other two, so the method used in the previous example can be applied to `while` and `for` loops as well.

- **Key Concept:** Though there are multiple ways of writing a loop in MIPS, conditional branch is key to decision making.
Inequalities in MIPS

- Until now, we’ve only tested equalities (== and != in C).
  - General programs need to test < and > as well.
- MIPS does not do branches on inequalities. Why?
- Create a MIPS Inequality Instruction:
  - “Set on Less Than”
  - Syntax: slt reg1, reg2, reg3
  - Meaning:
    ```
    if (reg2 < reg3)
    reg1 = 1;
    else reg1 = 0;
    ```
  - In general, “set” means “set to 1”, “reset” means “set to 0”.

UCSD
Inequalities in MIPS

- How do we use this?
- Compile by hand:
  
  ```
  if (g < h) goto Less;
  ```
- Use this mapping:
  
  ```
  g: $s0, h: $s1
  ```
Inequalities in MIPS

- Final compiled MIPS code:
  \[
  \text{slt } \$t0,\$s0,\$s1 \ # \ $t0 = 1 \text{ if } g<h \\
  \text{bne } \$t0,\$0,\text{Less} \ # \ \text{goto Less} \\
    \ # \ if \ $t0\neq0 \\
    \ # \ (if \ (g<h))
  \]

Less:

- **Branch if $t0 \neq 0 \Rightarrow (g < h)**
  - Register $0$ always contains the value $0$, so \texttt{bne} and \texttt{beq} often use it for comparison after an \texttt{slt} instruction.
Inequalities in MIPS

- Now, we can implement <, but how do we implement >, <= and >=?

- We could add 3 more instructions, but:
  - MIPS goal: Simpler is Better

- Can we implement <= in one or more instructions using just slt and the branches?

- What about >?

- What about >=?
Immediates in Inequalities

- There is also an immediate version of `slt` to test against constants: `slti`
  - Helpful in for loops

C

```c
if (g >= 1) goto Loop
```

Loop: . . .

MIPS

```mips
slti $t0,$s0,1  # $t0 = 1 if $s0<1 (g<1)
beq $t0,$0,Loop  # goto Loop # if $t0==0 # (if (g>=1))
```
What about unsigned numbers?

- there are unsigned inequality instructions: `sltu`, `sltiu`
- MSB='1' in signed numbers means a negative number

- `$s0 = FFFF FFFA_{hex}$, $s1 = 0000 FFFA_{hex}$
- What is value of $t0$, $t1$?
  - `slt $t0, $s0, $s1`
  - `sltu $t1, $s0, $s1`

\[
\begin{align*}
&s0 = FFFF FFFF \text{ (Hex)} \\
&s1 = 0000 0001 \text{ (Hex)} \\
\text{Sl}t \ t0, \ s0, \ s1 \\
\text{Sl}tu \ t1, \ s0, \ s1
\end{align*}
\]
Example: The C Switch Statement

Choose among four alternatives depending on whether $k$ has the value 0, 1, 2 or 3.

Compile this C code:

```c
switch (k) {
    case 0: f=i+j; break; /* k=0*/
    case 1: f=g+h; break; /* k=1*/
    case 2: f=g-h; break; /* k=2*/
    case 3: f=i-j; break; /* k=3*/
}
```
Example: The C Switch Statement

- This is complicated, so simplify.

- Rewrite it as a chain of if-else statements, which we already know how to compile:

  ```c
  if(k==0) f=i+j;
  else if(k==1) f=g+h;
  else if(k==2) f=g-h;
  else if(k==3) f=i-j;
  
  Use this mapping:
  f: $s0, g: $s1, h: $s2, i: $s3, j: $s4, k: $s5
  ```
Example: The C Switch Statement

```
bne $s5,$0,L1    # branch k! = 0
add $s0,$s3,$s4  # k==0 so f=i+j
j Exit # end of case so Exit
L1: addi $t0,$s5,-1  # $t0=k-1
    bne $t0,$0,L2    # branch k!=1
    add $s0,$s1,$s2  # k==1 so f=g+h
    j Exit # end of case so Exit
L2: addi $t0,$s5,-2  # $t0=k-2
    bne $t0,$0,L3    # branch k!=2
    sub $s0,$s1,$s2  # k==2 so f=g-h
    j Exit # end of case so Exit
L3: addi $t0,$s5,-3  # $t0=k-3
    bne $t0,$0,Exit  # branch k!=3
    sub $s0,$s3,$s4  # k==3 so f=i-j
Exit:
```

UCSD
Conclusion

- Instructions so far:
  - add, addi,
  - sub, subi,
  - mult, div,
  - lw, sw,
  - beq, bne, j,
  - slt, slti, sltu, sltiu

- Registers so far:
  - C Variables: $s0 - $s7
  - Temporary Variables: $t0 - $t9
  - Zero: $zero
Conclusions

- A Decision allows us to decide which pieces of code to execute at run-time rather than at compile-time.
- C Decisions are made using conditional statements within an if, while, do while or for.
- MIPS Decision making instructions are the conditional branches: beq and bne.
- In order to help the conditional branches make decisions concerning inequalities, we introduce a single instruction: “Set on Less Than” called slt, slti, sltu, sltui