EECE 321: Computer Organization

Mohammad M. Mansour
Dept. of Electrical and Compute Engineering
American University of Beirut

Lecture 8: MIPS ISA

Announcements

- Reading assignment
  - Sections 2.7, 2.8, 2.10, 2.13, 2.14
- Machine problem 1 due Friday March 12 @ 5:00pm
  - Submit on Moodle (one submission per team)
- Drop at the end of the lecture

Example

- Consider the following MIPS assembly code:

```
Loop:   addi $s0,$s0,-1          # i = i - 1
        addi $s1,$s1, 1          # j = j + 1
        slti $t0,$s1,2           # $t0 = (j < 2)
        beq  $t0,$0 ,Loop        # goto Loop if $t0 == 0 => j >= 2
        slt  $t0,$s1,$s0         # $t0 = (j < i)
        bne  $t0,$0 ,Loop        # goto Loop if $t0 != 0 => j < i
```

- Assume the following mapping:
  - `i:$s0, j:$s1`

- What C code properly fills in the blank in loop below?

```
do {
  i--;
  j++;
}
while ( _________ );
```

Summary

- In order to help the conditional branches make decisions concerning inequalities, we introduced a single instruction:
  - "Set on Less Than" called slt, slti, sltu, sltiu
- One can store and load (signed and unsigned) bytes as well as words
- Unsigned add/sub don't cause overflow
- New MIPS Instructions:

  sll, srl

  slt, slti, sltu, sltiu

  addu, addiu, subu

Procedures

C Functions

```
main() {
    int i, j, k, m;
    ...
    i = mult(j,k); ...
    m = mult(i,i); ...

}
```

```
/* Simple multiplication function */

int mult (int mcand, int mlier){
  int product;

  product = 0;
  while (mlier > 0){
    product = product + mcand;
    mlier = mlier – 1;

  }
    return product;
}
```

What information must the compiler/ programmer keep track of?

- Function call bookkeeping
  - Registers play a major role in keeping track of information for function calls.
  - Register conventions:
    - Return address: $ra
  - Arguments: $a0, $a1, $a2, $a3
  - Return value: $v0, $v1
  - Local variables: $s0, $s1, … , $s7
- What about
  - more arguments
  - more return values
  - Arrays, structures …? Use STACK.

Instruction Support for Functions

- Consider the following C function:

```
... sum(a,b);...          /* a : $s0, b : $s1 */
...
int sum(int x, int y) {
   return x + y;
}
```

- In MIPS:

```
address                Instruction
1000                   add  $a0,$s0,$zero     # x = a
1004                   add  $a1,$s1,$zero     # y = b
1008                   addi $ra,$zero,1016    # $ra = 1016
1012                   j    sum               # jump to sum
1016                   ...
2000        sum:       add  $v0,$a0,$a1
2004                   jr   $ra               # new instruction
```

- Why use `jr` here? Why not simply use `j`?
  - sum might be called by many functions, so we can't return to a fixed place. The calling procedure to sum must be able to say "return here" somehow.
- MIPS has a single instruction to jump and save return address:
  - jump and link (jal)

Instruction Support for Functions

Before:

```
1008 addi $ra,$zero,1016 #$ra=1016
1012 j sum                #go to sum
```

After:

```
1008 jal sum  # $ra=1012,go to sum
```

- Why have a jal?
  - Make the common case fast: function calls are very common.
  - Also, no need to know where the code is loaded into memory with jal.
- Syntax for jal (jump and link) is same as for j (jump):
  - jal label
- jal should really be called laj for "link and jump":
  - Step 1 (link): Save address of next instruction into $ra (Why next instruction?)
  - Step 2 (jump): Jump to the given label
- Syntax for jr (jump register): jr register
- Instead of providing a label to jump to, the jr instruction provides a register which contains an address to jump to.
- Only useful if we know exact address to jump to.
- Very useful for function calls:
  - jal stores return address in register ($ra)
  - jr $ra jumps back to that address

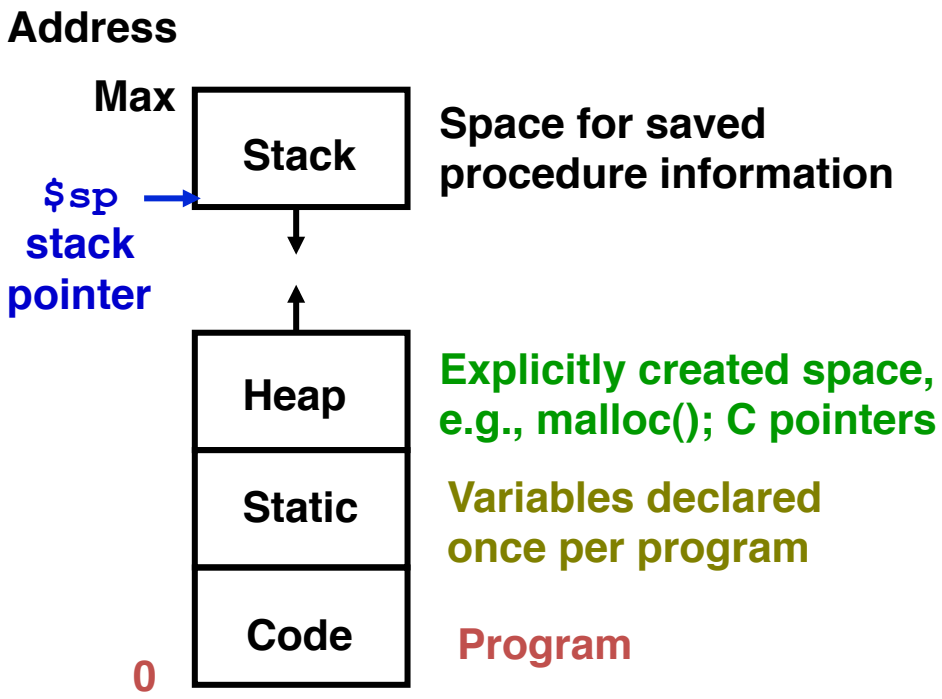
Nested Procedures

```
int sumSquare(int x, int y) {
        return mult(x,x)+ y;

}
```

- Function sumSquare calls function mult.
- So there's a value in $ra that sumSquare wants to jump back to, but this will be overwritten by the call to mult.
- Need to save sumSquare return address before the call to mult.
- In general, may need to save some other info in addition to $ra.
- When a C program is run, there are 3 important memory areas allocated:
  - *Static*: Variables declared once per program, cease to exist only after execution completes. E.g., C global variables
  - *Heap*: Variables declared dynamically
  - *Stack*: Space to be used by procedures during execution; this is where we can save register values

C Memory Allocation



- So we have a register $sp which always points to the last used space in the stack.
- To use stack, we decrement this pointer first by the amount of space we need and then fill it with info.
- So, how do we compile this?

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

Compilation Using the Stack

- Hand-compile

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

```
sumSquare:
"push"  addi $sp,$sp,-8          # space on stack
        sw $ra, 4($sp)           # save ret address
        sw $a1, 0($sp)           # save y

        add $a1,$a0,$zero        # prepare arguments for mult(x,x)
        jal mult                 # call mult

"pop"   lw $a1, 0($sp)           # restore y
        add $v0,$v0,$a1          # mult()+y

        lw $ra, 4($sp)           # get ret address

        addi $sp,$sp,8           # restore stack
        jr $ra

mult: ...                        # sets $v0
```

Steps for Making a Procedure Call

- Steps:
  1. Save all necessary values onto stack (using sw).
  2. Assign argument(s), if any.
  3. Use jal
  4. Restore values from stack (using lw).
- Rules for procedures:
  - Called with a jal instruction, returns with a jr $ra
  - Accepts up to 4 arguments in $a0,$a1,$a2,$a3
  - Return value is always in $v0 (and if necessary in $v1)
  - Must follow register conventions (even in functions that only you will call).
- Basic structure of a function:

Prologue

Body

Epilogue

```
entry_label:
addi $sp,$sp, -framesize   # allocate frame area
sw $ra, framesize-4($sp)   # save $ra
save other regs if need be

        ...                # (call other functions…)

restore other regs if need be
lw $ra, framesize-4($sp)   # restore $ra
addi $sp,$sp, framesize
jr $ra
```