Functions and Stacks

Lecture 7
CAP 3103
06-09-2014
Register Operands

- Arithmetic instructions use register operands
- MIPS has a $32 \times 32$-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0$, $t1$, …, $t9$ for temporary values
  - $s0$, $s1$, …, $s7$ for saved variables
- Design Principle 2: Smaller is faster
  - c.f. main memory: millions of locations
Register Operand Example

- C code:
  \[
  f = (g + h) - (i + j);
  \]
  - f, ..., j in $s0, ..., $s4

- Compiled MIPS code:
  
  ```
  add $t0, $s1, $s2  
  add $t1, $s3, $s4  
  sub $s0, $t0, $t1  
  ```
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
Memory Operand Example 1

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

- Compiled MIPS code:
  
  ```mips
  Index 8 requires offset of 32
  ```
  
  - 4 bytes per word
  
  ```mips
  lw  $t0, 32($s3)    # load word
  add $s1, $s2, $t0
  ```
MIPS R-format Instructions

Instruction fields
- **op**: operation code (opcode)
- **rs**: first source register number
- **rt**: second source register number
- **rd**: destination register number
- **shamt**: shift amount (000000 for now)
- **funct**: function code (extends opcode)
### MIPS I-format Instructions

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **Immediate arithmetic and load/store instructions**
  - rt: destination or source register number
  - Constant: $-2^{15}$ to $+2^{15} - 1$
  - Address: offset added to base address in rs
MIPS J-format Instructions

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- (Pseudo)Direct jump addressing
  - Target address = PC_{31...28} : (address \times 4)
Addressing Mode Summary

1. Immediate addressing
   \[ \text{op rs rt \quad Immediate} \]

2. Register addressing
   \[ \text{op rs rt rd \ldots funct} \]
   \[ \text{Registers} \]
   \[ \text{Register} \]

3. Base addressing
   \[ \text{op rs rt Address} \]
   \[ \text{Register} \]
   \[ \text{Memory} \]
   \[ \text{Byte} \quad \text{Halfword} \quad \text{Word} \]

4. PC-relative addressing
   \[ \text{op rs rt Address} \]
   \[ \text{PC} \]
   \[ \text{Memory} \]
   \[ \text{Word} \]

5. Pseudodirect addressing
   \[ \text{op Address} \]
   \[ \text{PC} \]
   \[ \text{Memory} \]
   \[ \text{Word} \]
C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)
  ```c
  void swap(int v[], int k)
  {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
  }
  ```
- v in $a0, k in $a1, temp in $t0
## The Procedure Swap

```assembly
swap:  sll $t1, $a1, 2  # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
      # (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```
C functions

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

/* really dumb mult function */
int mult (int mcand, int mlier){
    int product = 0;
    while (mlier > 0) {
        product = product + mcand;
        mlier = mlier -1; }
    return product;
}
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 $s0, s1, ..., s7$

- The stack is also used; more later.
... sum(a,b);... /* a,b:$s0,$s1 */
}

int sum(int x, int y) {
    return x+y;
}

address (shown in decimal)
1000
1004
1008
1012
1016
...
2000
2004

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.
... sum(a,b);... /* a,b:$s0,$s1 */
}

int sum(int x, int y) {
    return x+y;
}

address (shown in decimal)
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
Instruction Support for Functions (3/6)

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

• Question: Why use jr here? Why not use j?

• Answer: sum might be called by many places, so we can’t return to a fixed place. The calling proc to sum must be able to say “return here” somehow.

2000 sum: add $v0,$a0,$a1
2004 jr $ra # new instruction
Instruction Support for Functions (4/6)

- Single instruction to jump and save return address: jump and link (*jal*)

- Before:

  1008 addi $ra,$zero,1016  #$ra=1016
  1012 j sum                   #goto sum

- After:

  1008 jal sum  #$ra=1012,goto sum

- Why have a *jal*?
  - Make the common case fast: function calls very common.
  - Don’t have to know where code is in memory with *jal*!
Instruction Support for Functions (5/6)

- 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? Why not current one?
  - Step 2 (jump): Jump to the given label
Instruction Support for Functions (6/6)

- Syntax for jr (jump register):

  
  \[
  \text{jr register}
  \]

- Instead of providing a label to jump to, the jr instruction provides a register which contains an address to jump to.

- Very useful for function calls:
  - jal stores return address in register (\$ra)
  - jr \$ra jumps back to that address
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}

- Something called `sumSquare`, now `sumSquare` is calling `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 call to `mult`.
Nested Procedures (2/2)

- 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 globals
  - **Heap**: Variables declared dynamically via `malloc`
  - **Stack**: Space to be used by procedure during execution; this is where we can save register values
C Memory Allocation

- **Address**: Infinite
  - Stack
    - Space for local vars, saved procedure information
  - Heap
    - Explicitly created space, i.e., `malloc()`
  - Static
    - Variables declared once per program; e.g., globals (doesn’t change size)
  - Code
    - Program (doesn’t change size)

Dr Dan Garcia
main ()
{
    proc_1(1);
}

void proc_1 (int a)
{
    proc_2(2);
}

void proc_2 (int b)
{
    proc_3(3);
}

void proc_3 (int c)
{
}
Using the Stack (1/2)

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

```cpp
int sumSquare(int x, int y) {
    return mul(x, x) + y;
}
```
Using the Stack (2/2)

- **Hand-compile**

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

  sumSquare:

  addi $sp,$sp,-8  # space on stack
  sw $ra, 4($sp)  # save ret addr
  sw $a1, 0($sp)  # save y
  add $a1,$a0,$zero  # mult(x,x)
  jal mult  # call mult
  lw $a1, 0($sp)  # restore y
  add $v0,$v0,$a1  # mult()+y
  lw $ra, 4($sp)  # get ret addr
  addi $sp,$sp,8  # restore stack
  jr $ra

  mult: ...
The Sort Procedure in C

- Non-leaf (calls swap)

```c
void sort (int v[], int k) {
    int i, j;
    for (i = 0; i < k; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v, j);
        }
    }
}
```

- v in $a0, k in $a1, i in $s0, j in $s1
## The Procedure Body

```assembly
move $s2, $a0           # save $a0 into $s2
move $s3, $a1           # save $a1 into $s3
move $s0, $zero        # i = 0
for1tst: slt $t0, $s0, $s3   # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
    beq $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
    addi $s1, $s0, -1    # j = i - 1
for2tst: slti $t0, $s1, 0   # $t0 = 1 if $s1 < 0 (j < 0)
    bne $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
    sll $t1, $s1, 2     # $t1 = j * 4
    add $t2, $s2, $t1    # $t2 = v + (j * 4)
    lw  $t3, 0($t2)     # $t3 = v[j]
    lw  $t4, 4($t2)     # $t4 = v[j + 1]
    slt $t0, $t4, $t3   # $t0 = 0 if $t4 ≥ $t3
    beq $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
move $a0, $s2           # 1st param of swap is v (old $a0)
move $a1, $s1           # 2nd param of swap is j
jal swap                 # call swap procedure
addi $s1, $s1, -1       # j -= 1
j for2tst               # jump to test of inner loop
exit2: addi $s0, $s0, 1  # i += 1
j for1tst               # jump to test of outer loop
```

---

### Inner loop
- for2tst: slti, bne
- sll, add, lw, lw, slt, beq

### Outer loop
- for1tst: slt, beq
- addi, jal, addi, j
## The Full Procedure

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $sp,$sp, -20</td>
<td>make room on stack for 5 registers</td>
</tr>
<tr>
<td>sw $ra, 16($sp)</td>
<td>save $ra on stack</td>
</tr>
<tr>
<td>sw $s3,12($sp)</td>
<td>save $s3 on stack</td>
</tr>
<tr>
<td>sw $s2, 8($sp)</td>
<td>save $s2 on stack</td>
</tr>
<tr>
<td>sw $s1, 4($sp)</td>
<td>save $s1 on stack</td>
</tr>
<tr>
<td>sw $s0, 0($sp)</td>
<td>save $s0 on stack</td>
</tr>
</tbody>
</table>

...  # procedure body

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $s0, 0($sp)</td>
<td>restore $s0 from stack</td>
</tr>
<tr>
<td>lw $s1, 4($sp)</td>
<td>restore $s1 from stack</td>
</tr>
<tr>
<td>lw $s2, 8($sp)</td>
<td>restore $s2 from stack</td>
</tr>
<tr>
<td>lw $s3,12($sp)</td>
<td>restore $s3 from stack</td>
</tr>
<tr>
<td>lw $ra,16($sp)</td>
<td>restore $ra from stack</td>
</tr>
<tr>
<td>addi $sp,$sp, 20</td>
<td>restore stack pointer</td>
</tr>
<tr>
<td>jr $ra</td>
<td>return to calling routine</td>
</tr>
</tbody>
</table>
```
Steps for Making a Procedure Call

1. Save necessary values onto stack.
2. Assign argument(s), if any.
3. jal call
4. Restore values from stack.
Rules for Procedures

- Called with a `jal` instruction, returns with a `jr $ra`
- Accepts up to 4 arguments in `$a0, $a1, $a2` and `$a3`
- Return value is always in `$v0` (and if necessary in `$v1`)
- Must follow register conventions

So what are they?
Basic Structure of a Function

**Prologue**

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

**Body** . . . (call other functions…)

**Epilogue**

restore other regs if need be
lw $ra, framesize-4($sp)  # restore $ra
addi $sp,$sp, framesize
jr $ra
<table>
<thead>
<tr>
<th><strong>MIPS Registers</strong></th>
<th><strong>The constant 0</strong></th>
<th>$0</th>
<th>$zero</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reserved for Assembler</strong></td>
<td>$1</td>
<td>$at</td>
<td></td>
</tr>
<tr>
<td><strong>Return Values</strong></td>
<td>$2-$3</td>
<td>$v0-$v1</td>
<td></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>$4-$7</td>
<td>$a0-$a3</td>
<td></td>
</tr>
<tr>
<td><strong>Temporary</strong></td>
<td>$8-$15</td>
<td>$t0-$t7</td>
<td></td>
</tr>
<tr>
<td><strong>Saved</strong></td>
<td>$16-$23</td>
<td>$s0-$s7</td>
<td></td>
</tr>
<tr>
<td><strong>More Temporary</strong></td>
<td>$24-$25</td>
<td>$t8-$t9</td>
<td></td>
</tr>
<tr>
<td><strong>Used by Kernel</strong></td>
<td>$26-27</td>
<td>$k0-$k1</td>
<td></td>
</tr>
<tr>
<td><strong>Global Pointer</strong></td>
<td>$28</td>
<td>$gp</td>
<td></td>
</tr>
<tr>
<td><strong>Stack Pointer</strong></td>
<td>$29</td>
<td>$sp</td>
<td></td>
</tr>
<tr>
<td><strong>Frame Pointer</strong></td>
<td>$30</td>
<td>$fp</td>
<td></td>
</tr>
<tr>
<td><strong>Return Address</strong></td>
<td>$31</td>
<td>$ra</td>
<td></td>
</tr>
</tbody>
</table>
“And in Conclusion…”

- Functions called with `jal`, return with `jr $ra`.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far…
  Arithmetic: `add`, `addi`, `sub`, `addu`, `addiu`, `subu`
  Memory: `lw`, `sw`, `lb`, `sb`
  Decision: `beq`, `bne`, `slt`, `slti`, `sltu`, `sltiu`
  Unconditional Branches (Jumps): `j`, `jal`, `jr`

- Registers we know so far
  - All of them!
Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address

- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity
### Example: Clearing and Array

| clear1(int array[], int size) { | clear2(int *array, int size) { |
| int i; | int *p; |
| for (i = 0; i < size; i += 1) | for (p = &array[0]; p < &array[size]; |
| array[i] = 0; | p = p + 1) |
| *p = 0; | *p = 0; |

| move $t0,$zero # i = 0 | move $t0,$a0 # p = & array[0] |
| loop1: sll $t1,$t0,2 # $t1 = i * 4 | sll $t1,$a1,2 # $t1 = size * 4 |
| add $t2,$a0,$t1 # $t2 = | add $t2,$a0,$t1 # $t2 = |
| # &array[i] | # &array[size] |
| sw $zero, 0($t2) # array[i] = 0 | loop2: sw $zero,0($t0) # Memory[p] = 0 |
| addi $t0,$t0,1 # i = i + 1 | addi $t0,$t0,4 # p = p + 4 |
| slt $t3,$t0,$a1 # $t3 = | slt $t3,$t0,$t2 # $t3 = |
| # (i < size) | #(p<&array[size]) |
| bne $t3,$zero,loop1 # if (...) | bne $t3,$zero,loop2 # if (...) |
| # goto loop1 | # goto loop2 |
Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer
Who cares about stack management?

- Pointers in C allow access to deallocated memory, leading to hard-to-find bugs!

```c
int *ptr () {
    int y;
y = 3;
    return &y;
}
main () {
    int *stackAddr, content;
    stackAddr = ptr();
    content = *stackAddr;
    printf("%d", content);  /* 3 */
    content = *stackAddr;
    printf("%d", content);  }/!*13451514 */
```
Memory Management

- How do we manage memory?
- Code, Static storage are easy: they never grow or shrink
- Stack space is also easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- Managing the heap is tricky: memory can be allocated / deallocated at any time
Heap Management Requirements

- Want `malloc()` and `free()` to run quickly.
- Want minimal memory overhead
- Want to avoid fragmentation* – when most of our free memory is in many small chunks
  - In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

* This is technically called external fragmentation
Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes
Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
    - Memory has become fragmented!
  - We have to keep track of the two freespace regions
  - Request R3 for 50 bytes
    - We have to search the data structures holding the freespace to find one that will fit! Choice here...
Register Conventions (1/4)

- **CalleR**: the calling function
- **CalleE**: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- **Register Conventions**: A set of generally accepted rules as to which registers will be unchanged after a procedure call (jal) and which may be changed.
Register Conventions (2/4) – saved

- $0: No Change. Always 0.
- $s0 – s7: Restore if you change. Very important, that’s why they’re called saved registers. If the callee changes these in any way, it must restore the original values before returning.
- $sp: Restore if you change. The stack pointer must point to the same place before and after the jal call, or else the caller won’t be able to restore values from the stack.
- HINT -- All saved registers start with S
Register Conventions (2/4) – volatile

- $ra: Can Change. The jal call itself will change this register. Caller needs to save on stack if nested call.
- $v0 – $v1: Can Change. These will contain the new returned values.
- $a0 – $a3: Can change. These are volatile argument registers. Caller needs to save if they are needed after the call.
- $t0 – $t9: Can change. That’s why they’re called temporary: any procedure may change them at any time. Caller needs to save if they’ll need them afterwards.
What do these conventions mean?

- If function R calls function E, then function R must save any temporary registers that it may be using onto the stack before making a jal call.
- Function E must save any S (saved) registers it intends to use before garbling up their values, and restore them after done garbling.

Remember: caller/callee need to save only temporary/saved registers they are using, not all registers.
Question?

What does \( r \) have to push on the stack before “jal e”?

a) 1 of \( \{s0, sp, v0, t0, a0, ra\} \)
b) 2 of \( \{s0, sp, v0, t0, a0, ra\} \)
c) 3 of \( \{s0, sp, v0, t0, a0, ra\} \)
d) 4 of \( \{s0, sp, v0, t0, a0, ra\} \)
e) 5 of \( \{s0, sp, v0, t0, a0, ra\} \)