EECS 314 Computer Architecture

Language of the Machine

Function Calling
Review: Control Flow

- 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
Review: Control flow: if, ?:, while, for

- if (condition) s1; else s2;
  
  if (!condition) goto L1;
  s1;
  goto L2;

  L1: s2; /* else */
  L2:

- while (condition) s1;

  L2: if (!condition) goto L1;
  s1;
  goto L2;

  L1: /* exit loop */

- variable = condition ? s1 : s2;

  if (!condition) goto L1;
  variable=s1;
  goto L2;

  L1: variable=s2; /* else */
  L2:

- for (init; condition; inc) s1;

  init;
  L2: if (!condition) goto L1;
  s1;
  inc;
  goto L2;

  L1: /* exit loop */
Control flow: do-while

- `while (condition) s1;`
- `for(;condition; ) s1;`
- `do s1; while (condition);`
- `for(s1;condition; ) s1;`

L2: if (! condition) goto L1;
    s1;
    goto L2;

L1: /* exit loop */

L2:
    s1;
    if (condition) goto L2;

/* exit loop by fall through */

- Tests the termination condition at the top.
- Tests the termination condition at the bottom after making each pass through the loop body.

- 0 or more times
- 1 or more times
Control flow: break (from K&R)

- A **break** causes the **innermost** enclosing loop or switch to be exited immediately.

```c
/* clear lower triangle array */
for(i=0; i<10; i++) {
    for(j=0; j<10; j++) {
        if (i >= j) break;
        a[i][j] = 0;
    }
}
```

```c
i=0;
L2: if (i >= 10) goto L1;
    j=0;
L4: if (j >= 10) goto L3;
    if (i >= j) goto L3;
    a[i][j] = 0;
    j++;
    goto L4;
L3: /* exit loop */
    i++;
    goto L2;
L1: /* exit loop */
```
MIPS Goto Instruction

- In addition to conditional branches, MIPS has an unconditional branch: \textit{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): \textit{goto label}

- Technically, it’s the same as: \textit{beq $0,$0,label}

- since it always satisfies the condition.
Control Machine Instructions

- **beq**  $rs, $rt, wordoffset16  
  $$\text{if } (rs = = rt) \text{ goto wordoffset16;}$$
- **bne**  $rs, $rt, wordoffset16  
  $$\text{if } (rs ! = rt) \text{ goto wordoffset16;}$$
- **j**  wordoffset26  
  $$\text{goto wordoffset26;}$$

- **slt**  $rd, $rs, $rt  
  $$rd = (rs < rt) ? 1 : 0;$$
- **slti**  $rt, $rs, const16  
  $$rt = (rs < \text{const16}) ? 1 : 0;$$
- **sltu**  $rd, $rs, $rt  
  $$rd = ((\text{unsigned})rs < (\text{unsigned})rt) ? 1 : 0;$$
- **sltiu**  $rt, $rs, const16  
  $$rt = ((\text{unsigned})rs < \text{const16}) ? 1 : 0;$$
Structured programming (Programming Languages, K. Louden)

• Ever since a famous letter by E. W. Dijkstra in 1968, GOTOs have been considered suspect, since

• they can so easily lead to unreadable “spaghetti” code.

• The GOTO statement is very close to actual machine code.

• As Dijkstra pointed out, its “unbridled” use can compromise even the most careful language design and lead to undecipherable programs.

• Dijkstra proposed that its use be severely controlled or even abolished.

• This unleashed one of the most persistent controversies in programming, which still rages today...
Structured programming (Programming Languages, K. Louden)

- **efficiency:** One group argues that the GOTO is indispensable for efficiency & even for good structure.
  - Such as state machines (LEX, YACC, parsers)
  - Break out of deeply nested loop in one step
    - C/C++ can only do inner most loop
    - C/C++ can use exit flags in each loop level (ugly)
  - GOTOs should only jump forward (never backward)
  - Error handling (gotos are still more efficient)
    - C/C++/Unix can use the signal( ) function
    - C++ can use the throw/catch statements

- **limited:** Another argues that it can be useful under carefully limited circumstances. (parsers, state machines).

- **abolish:** A third argues that it is an anachronism that should truly be abolished henceforth from all computer languages.
Control flow: continue (from K&R)

- The `continue` statement is related to the break. C/C++ is one of the few languages to have this feature.
- It causes the next iteration of the enclosing for, while, or do loop to begin.
- In the `while` and `do`, this means that the condition part is executed immediately.
- In the `for`, control passes to the increment step.

```c
/* abs(array) */
for(i=0; i < n; i++) {
    if (a[i] > 0) continue;
    a[i] = -a[i];
}
i=0;
L2: if (i >= n) goto L1;
   if (a[i] > 0) goto L2c;
   a[i] = -a[i];
L2c: i++;
goto L2;
L1:
```
Logical Operators: && and | |

(From K&R)

• More interesting are the logical operators && and | |

• Bitwise and ( & ), bitwise or ( | ), bitwise not ( ~ )
  – Bitwise operators imply no order and parallel in nature

• Logical and ( && ), logical or( || ), logical not ( ! )
  – Logical operators imply order and sequential in nature

• Expressions connected by && and | | are evaluated left to right, and

• evaluation stops as soon as the truth or falsehood of the result is know.

• Most C programs rely on the above properties:
  (1) left to right evaluation (2) stop as soon as possible.
Logical Operators: example

- For example, here is a loop from the input function getline

```c
for(i=0; i<limit-1 && (c=getchar())!='\n' && c!=EOF ; i++ ) {
    a[i] = c;
}
```

- Before reading a new character it is necessary to check that there is room to store it in the array `a`.
- So the test `i<limit-1` must be made first
- Moreover, if the test fails, we must not go on and read another character

```
i=0;
L2: if (i >= limit-1) goto L1;
c=getchar();
if (c == '\n') goto L1;
if (c == EOF) goto L1;
a[i] = c;
i++;
goto L2;
L1:
```
Review: slti example

- C code fragment
  
  ```
  if (i < 20) { f=g+h; }
  else { f=g-h; }
  ```

- Re-written C code
  
  ```
  temp = (i < 20)? 1 : 0;
  if (temp == 0) goto L1;
  f=g+h;
  goto L2;
  L1:
  f=g-h;
  L2:
  ```

- MIPS code
  
  ```
  slti $t1,$s3,20
  beq $t1,$0,L1
  add $s0,$s1,$s2
  j L2
  L1:
  sub $s0,$s1,$s2
  L2:
  ```
C functions

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

int mult (int x, int y) {
    int f;
    for (f= 0; y > 0; y--) {
        f += x;
    }
    return f;
}

- Functions, procedures one of main ways to give a program structure, and encourage reuse of code.
- But they do not add any more computational power.

What information must compiler/programmer keep track of?
Calling functions: Bookkeeping

- Function address
- Return address $ra$ (same as $31$)
- Arguments $a0, a1, a2, a3$
- Return value $v0, v1$
- Local variables $s0, s1, ..., s7$

Most problems above are solved simply by using register conventions.
Calling functions: example

... c=sum(a,b); ... /* a,b,c:$s0,$s1,$s2 */

}

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

address

1000  add  $a0,$s0,$0  # x = a
1004  add  $a1,$s1,$0  # y = b
1008  addi $ra,$0,1016  # $ra=1016
1012  j      sum  # jump to sum
1016  add  $s2,$0,$v0  # c=$v0

... Why jr $ra vs. j 1016 to return?

2000  sum: add $v0,$a0,$a1  # x+y
2004  jr  $ra  # pc = $ra = 1016
Calling functions: jal, jump and link

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

- slow way:
  1008  addi  $ra,$zero,1016  #$ra=1016
  1012  j       sum  #go to sum

- faster way and save one instruction:
  1012  jal  sum  # pc = $ra = 1016

- but adds more complexity to the hardware

- Why have a jal? Make the common case fast: functions are very common.
Calling functions: setting the return address

• Syntax for jal (jump and link) is same as for j (jump):

\[
\text{jal label} \quad \# \text{ reg[\$ra]=pc+4; pc=label}
\]

• jal should really be called laj for “link and jump”:

• Step 1 (link):
  \text{Save address of next instruction into \$ra} (Why?)

• Step 2 (jump):
  \text{Jump to the given label}
Calling functions: return

• Syntax for jr (jump register):

\[
\text{jr} \quad \$\text{register} \quad \# \ \text{reg}[$\text{pc}$] = \$\text{register}
\]

• Instead of providing a label to jump to, the \textit{jr} instruction provides a \texttt{register} that contains an address to jump to.

• Usually used in conjunction with jal, to jump back to the address that \texttt{jal} stored in $\$\text{ra}$ before function call.
Calling nested functions: example

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

- Something called sumSquare, now `sumSquare` is calling `mult(x, x)`.
- 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(x, x)`.
Calling nested functions: memory areas

- 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
  - **Heap:** Variables declared dynamically
  - **Stack:** Space to be used by procedure during execution; this is where we can save register values

  - *Not* identical to the “stack” data structure!
C memory Allocation

Address
∞

$sp
stack
pointer

0

Space for saved procedure information

Stack

Heap

Static

Code

Explicitly created space, e.g., malloc(); C pointers

Variables declared once per program (.data segment)

Program (.text segment)
Stack Discipline

- C, C++, Java follow “Stack Discipline”;
  - e.g., D cannot return to A bypassing B
  - Frames can be adjacent in memory
  - Frames can be allocated, discarded as a LIFO (stack)

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

sumSquare:

Prologue

    subi $sp, $sp,12
    sw $ra,  8($sp)
    sw $a1, 4($sp)
    sw $a0, 0($sp)

Body

    addi $a1, $a0,$0
    jal mult

    lw $a0,0($sp)
    lw $a1,4($sp)

Epilogue

    lw $ra, 8($sp)
    add $v0,$v0,$a1
    addi $sp,$sp,12
    jr $ra

# push stack stack
# push return addr
# push y
# push x
# mult(x,x)
# call mult
# pop x
# pop y
# pop return addr
# mult()+y
# pop stack space
Frame Pointer

- The $fp$ points to the first word of the frame of a function.

- A $sp$ might change during a function and so references to a local variable in memory might have different offsets depending where they are in the function, making it harder to understand.

```c
int f(int x, int y) {
    int i, a=4, f;
    for(i=0;i<10;i++) {
        int a[20];
        if (!i) { a[0]=x; } else { a[i]=a[i-1]+y; }
        f=a[i];
    }
}
```
Memory Allocation

- C Procedure Call Frame
- Pass arguments ($a0-$a3 )
- Save caller-saved regs
- call function: jal
- space on stack ($sp-n)
  $sp@last word of frame
- Save callee-saved regs
- set $fp ($sp+n-4)
  $fp@first word of frame

Address  
 stack  
grows  

<table>
<thead>
<tr>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>$fp</td>
</tr>
<tr>
<td>saved $a0-a3</td>
</tr>
<tr>
<td>$ra</td>
</tr>
<tr>
<td>Saved Registers</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Local Variables</td>
</tr>
</tbody>
</table>

low
### MIPS Register Summary

<table>
<thead>
<tr>
<th>Registers</th>
<th>Total Regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Zero, $0</td>
<td>1</td>
</tr>
<tr>
<td>(Return) Value registers ($v0,$v1)</td>
<td>3</td>
</tr>
<tr>
<td>Argument registers ($a0-$a3)</td>
<td>7</td>
</tr>
<tr>
<td>Return Address ($ra)</td>
<td>8</td>
</tr>
<tr>
<td>Saved registers ($s0-$s7)</td>
<td>16</td>
</tr>
<tr>
<td>Temporary registers ($t0-$t9)</td>
<td>26</td>
</tr>
<tr>
<td>Global Pointer ($gp)</td>
<td>27</td>
</tr>
<tr>
<td>Stack Pointer ($sp)</td>
<td>28</td>
</tr>
<tr>
<td>Frame Pointer ($fp), or $t10</td>
<td>29</td>
</tr>
</tbody>
</table>

- 2 for OS ($k0, $k1), 1 for assembler ($at)