Announcements

• Project 2 out
  • Due Thu Oct 4 (7 days)

• Project 3 out
  • Due Tuesday Oct 9 (12 days)

• Reminder: CS Nights Mondays 8pm-10pm, in MD119. Pizza provided!
Today

- Function calls
  - Calling convention
  - How to implement functions
Extending Fish

Let’s extend Fish with functions and local variables

```haskell
type exp = ...
    | Call of var * (exp list)

type stmt = ...
    | Let of var*exp*stmt

type func = { name : var, args : var list,
             body : stmt }

type prog = func list
```

One distinguished function (main) will be the entry point for the program
Call and Return

• Each procedure is just a Fish program beginning with a label (the function name)

• MIPS calling convention:
  • To compile a call \( f(a, b, c, d) \)
    • Move results of expressions \( a, b, c, d \) into registers \( $4–$7 \)
    • \texttt{jal } \( f \): moves \textbf{return address} into \( $31 \)
      ‣ The return address is address to continue execution after \( f \) has finished executing
      ‣ i.e., instruction immediately after the \texttt{jal}: \( \text{pc} + 4 \)
  
• To \texttt{return(e)}
  • Move result of \( e \) to \( $2 \)
  • \texttt{jr } $31 \text{ (i.e., jump to the return address)}
What Could Go Wrong?

• What if \texttt{foo} calls \texttt{bar} and \texttt{bar} calls \texttt{baz}?
  • \texttt{bar} needs to save its return address
    • $31$ is a \texttt{caller-save} register
  • Where do we save it?
  • One option: each procedure has a (global) variable to hold the return address
    • E.g., \texttt{foo\_return}, \texttt{bar\_return}, \texttt{baz\_return}
• But what about recursive calls? E.g., \texttt{foo} calls \texttt{bar}, and \texttt{bar} calls \texttt{foo}, and \texttt{foo} calls \texttt{bar}, ...
  • Each invocation of a function needs its own return address!
  • Each invocation of function also needs its own local variables, arguments, etc.
Stacks

• Key idea: associate a frame with each invocation of a procedure
  • In the frame, store data belonging to the invocation
    • Return address
    • Arguments to invocation
    • Local variables
    • ...

Memory

Frame for 1st invocation of foo

Frame for 1st invocation of bar
Frame Allocation

- Frames are allocated last-in-first-out
  - i.e., as a stack
  - For historic reasons, the stack of frames grows downwards
    - Why?
- We use $29$ (aka $sp$) as the stack pointer
  - Points to the top of the stack
- Use register $30$ (aka $fp$) as the frame pointer
  - Points to start of current frame (the first word in current frame)
Frame Allocation

- To allocate a frame with $n$ bytes:
  - Subtract $n$ from $sp$
  - Set $fp$ to $sp + n - 4$
    - i.e., $fp$ points to first word in this frame
- To deallocate a frame
  - Restore $fp$ to previous value
  - Add $n$ to $sp$
Calling Convention in More Detail

• To call $f$ with arguments $a_1$, ..., $a_n$:

1. Save **caller-save** registers
   • These are registers that the callee $f$ is free to clobber, so if the caller wants to preserve their values, caller must save them
   • Registers $8–15$, $24$, $25$ (aka $t0–t9$) are the general-purpose caller-save registers

2. Move arguments
   • Push extra arguments onto stack in **reverse order**
   • Place 1st four arguments in $4–7$ (aka $a0–a3$)
   • Set aside space on stack for 1st 4 arguments

3. Execute **jal** $f$: return address is placed in $31$ (aka $ra$)
   • [code for function $f$ executes, and returns to return address]

4. Pop arguments off stack; restore caller-save registers
What does the callee f do?

• Function prologue
  • At beginning of called function

• During execution

• Function epilogue
  • At end of called function
Function Prologue

1. Allocate frame: subtract frame size $n$ from $sp$
   - $n$ big enough for local vars, callee-save registers, etc.

2. Save any **callee-save registers**
   - Registers the caller expects to be preserved
   - Includes $fp$, $ra$, and $s0$–$s7$ ($16$–$23$).
   - Don't need to save a register you don't clobber…
     - E.g., only need to save $ra$ if function makes a call

3. Set $fp$ to $sp + n - 4$
During Execution

• Access variables relative to stack pointer (or frame pointer)
  • must keep track of each var's offset
• Temporary values can be pushed on stack and then popped off.

• Push(r): subu $sp,$sp,4; sw r, 0($sp)
• Pop(r): lw r,0($sp); addu $sp,$sp,4
• e.g., when compiling e1+e2, we can evaluate e1, push result on stack, evaluate e2, pop e1's value and then add the results.
Function Epilogue

1. Place result in $v0 ($2).
2. Restore callee-saved registers
   • Includes caller's frame pointer and the return address
3. Deallocate frame: add frame size $n$ to $\text{sp}$
4. Return to caller
   • \text{jr} $\text{ra}$
int fact(int n) {
    if (n < 1) return 1;
    else return n * fact(n-1);
}

int main() {
    return fact(10)+42;
}
Function prologue

main: subu $sp,$sp,32          # allocate frame
sw $ra,20($sp)                # save caller return address
sw $fp,16($sp)               # save caller frame pointer
addiu $fp,$sp,28             # set up new frame pointer
li $a0,10                   # set up argument (10)
jal fact                   # call fact
addi $v0,$v0,42             # add 42 to result
lw $ra,20($sp)              # restore return address
lw $fp,16($sp)              # restore frame pointer
addiu $sp,$sp,32            # pop frame
jr $ra                     # return to caller

Function epilogue
### Function prologue

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>li $a0, 10</td>
<td>set up argument (10)</td>
</tr>
<tr>
<td>jal fact</td>
<td>call fact</td>
</tr>
<tr>
<td>addi $v0, v0, 42</td>
<td>add 42 to result</td>
</tr>
</tbody>
</table>

### Function epilogue

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $ra, 20($sp)</td>
<td>restore return address</td>
</tr>
<tr>
<td>lw $fp, 16($sp)</td>
<td>restore frame pointer</td>
</tr>
<tr>
<td>addiu $sp, $sp, 32</td>
<td>pop frame</td>
</tr>
<tr>
<td>jr $ra</td>
<td>return to caller</td>
</tr>
</tbody>
</table>

### Notes:

- $sp is kept double-word aligned
- MIPS calling convention is minimum frame size is 24 bytes ($a0-$a3, $ra, padded to double-word boundary)
- $sp is kept double-word aligned
- main also needs to store $fp, padded to double-word boundary
- So frame size for main is 32 bytes
- save caller return address
- save caller frame pointer
- allocate frame
- subu $sp, $sp, 32  
- sw $ra, 20($sp)  
- sw $fp, 16($sp)
Fact

```assembly
Fact: subu $sp,$sp,32  # allocate frame
sw $ra,20($sp)  # save caller return address
sw $fp,16($sp)  # save caller frame pointer
addiu $fp,$sp,28  # set up new frame pointer

bgtz $a0,L2  # if n > 0 goto L2
li $v0,1  # set return value to 1
j L1  # goto epilogue

L2:
sw $a0,0($fp)  # save n
addi $a0,$a0,-1  # subtract 1 from n
jal fact  # call fact(n-1)
lw $v1,0($fp)  # load n
mul $v0,$v0,$v1  # calculate n*fact(n-1)

L1:
lw $ra,20($sp)  # restore ra
lw $fp,16($sp)  # restore frame pointer
addiu $sp,$sp,32  # pop frame from stack
jr $ra  # return
```

Function prologue

Function epilogue
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>saved argument 10</td>
</tr>
<tr>
<td>0x0FC</td>
<td>(filler to align to multiple of 8)</td>
</tr>
<tr>
<td>0x0F8</td>
<td>main’s return address</td>
</tr>
<tr>
<td>0x0F4</td>
<td>main’s frame pointer</td>
</tr>
<tr>
<td>0x0EC</td>
<td>(space for $a3)</td>
</tr>
<tr>
<td>0x0E8</td>
<td>(space for $a2)</td>
</tr>
<tr>
<td>0x0E4</td>
<td>(space for $a1)</td>
</tr>
<tr>
<td>0x0E0</td>
<td>(space for $a0)</td>
</tr>
</tbody>
</table>

Frame for fact(10)
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>saved argument 10</td>
</tr>
<tr>
<td>0x0FC</td>
<td>filler to align to multiple of 8</td>
</tr>
<tr>
<td>0x0F8</td>
<td>main’s return address</td>
</tr>
<tr>
<td>0x0F4</td>
<td>main’s frame pointer</td>
</tr>
<tr>
<td>0x0EC</td>
<td>(space for $a3)</td>
</tr>
<tr>
<td>0x0E8</td>
<td>(space for $a2)</td>
</tr>
<tr>
<td>0x0E4</td>
<td>(space for $a1)</td>
</tr>
<tr>
<td>0x0E0</td>
<td>(space for $a0)</td>
</tr>
<tr>
<td>0x0DC</td>
<td>saved argument 9</td>
</tr>
<tr>
<td>0x0D8</td>
<td>filler to align to multiple of 8</td>
</tr>
<tr>
<td>0x0D4</td>
<td>fact(10)’s return address</td>
</tr>
<tr>
<td>0x0D0</td>
<td>fact(10)’s frame pointer</td>
</tr>
<tr>
<td>0x0CC</td>
<td>(space for $a3)</td>
</tr>
<tr>
<td>0x0C8</td>
<td>(space for $a2)</td>
</tr>
<tr>
<td>0x0C4</td>
<td>(space for $a1)</td>
</tr>
<tr>
<td>0x0C0</td>
<td>(space for $a0)</td>
</tr>
<tr>
<td>0x0BC</td>
<td></td>
</tr>
<tr>
<td>0x0B8</td>
<td></td>
</tr>
<tr>
<td>0x0B4</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>0x100</td>
<td>saved argument 10</td>
</tr>
<tr>
<td>0x0FC</td>
<td>(filler to align to multiple of 8)</td>
</tr>
<tr>
<td>0x0F8</td>
<td>main’s return address</td>
</tr>
<tr>
<td>0x0F4</td>
<td>main’s frame pointer</td>
</tr>
<tr>
<td>0x0F0</td>
<td></td>
</tr>
<tr>
<td>0x0EC</td>
<td>(space for $a3)</td>
</tr>
<tr>
<td>0x0E8</td>
<td>(space for $a2)</td>
</tr>
<tr>
<td>0x0E4</td>
<td>(space for $a1)</td>
</tr>
<tr>
<td>0x0E0</td>
<td>(space for $a0)</td>
</tr>
<tr>
<td>0x0DC</td>
<td>saved argument 9</td>
</tr>
<tr>
<td>0x0D8</td>
<td>(filler to align to multiple of 8)</td>
</tr>
<tr>
<td>0x0D4</td>
<td>fact(10)'s return address</td>
</tr>
<tr>
<td>0x0D0</td>
<td>fact(10)'s frame pointer</td>
</tr>
<tr>
<td>0x0CC</td>
<td>(space for $a3)</td>
</tr>
<tr>
<td>0x0C8</td>
<td>(space for $a2)</td>
</tr>
<tr>
<td>0x0C4</td>
<td>(space for $a1)</td>
</tr>
<tr>
<td>0x0C0</td>
<td>(space for $a0)</td>
</tr>
<tr>
<td>0x0BC</td>
<td></td>
</tr>
<tr>
<td>0x0B8</td>
<td></td>
</tr>
<tr>
<td>0x0B4</td>
<td></td>
</tr>
</tbody>
</table>
WTFrame Pointer?

• Frame pointers aren't necessary!
  • Can calculate variable offsets relative to $sp$
  • Works unless values of unknown size are allocated on the stack (e.g., via `alloca`)
  • Simplifies variadic functions
    • i.e., variable number of arguments, such as `printf`
  • Debuggers like having saved frame pointers around (can crawl up the stack).

• There are 2 conventions for the MIPS:
  • GCC: uses frame pointer
  • SGI: doesn't use frame pointer

• No frame pointer means fewer instructions for calls, but complicates code generation (due to pushes, variadic functions, etc.)
Compiling Functions

• Now we understand calling convention, how do we generate code for functions?

• Must generate prologue & epilogue
  • Need to know how much space frame occupies
  • Roughly $c + 4\times v$ where $c$ is the constant overhead to save callee-save registers and $v$ is number of local variables (including parameters)

• When translating the body, must know offset of each variable
  • During compilation keep an environment that maps variables to offsets.
  • Access variables relative to the frame pointer.

• When we encounter a return, move the result to $v0$ and jump to epilogue
  • Also keep epilogue's label in environment
Environments

type varmap
val empty_varmap : unit -> varmap
val insert_var : varmap -> var -> int -> varmap
val lookup_var : varmap -> var -> int

type env = {epilogue : label,
              varmap : varmap}
What about temps?

• Three different options
  • For Project 4, implement one of these options

• Option 1
  • When evaluating a compound expression $e_1 + e_2$
    • generate code to evaluate $e_1$ and place it in $v0$, then push $v0$ on the stack
    • generate code to evaluate $e_2$ and place it in $v0$
    • pop $e_1$'s value into a temporary register (e.g., $t0$)
    • add $t0$ and $v0$ and put the result in $v0$
  • Bad news: lots of pushes and pops, so lots of overhead
  • Good news: very simple! Don’t need to figure out how many temps you need
Option 1 Example: 20 instructions
(11 memory)

\[ a := (x + y) + (z + w) \]

lw $v0, <xoff>($fp)    # evaluate x
push $v0               # push x's value
lw $v0, <yoff>($fp)    # evaluate y
pop $v1                # pop x's value
add $v0,$v1,$v0        # add x and y's values
push $v0               # push value of x+y
lw $v0, <zoff>($fp)    # evaluate z
push $v0               # push z's value
lw $v0, <woff>($fp)    # evaluate w
pop $v1                # pop z's value
add $v0,$v1,$v0        # add z and w's values
pop $v1                # pop x+y
add $v0,$v1,$v0        # add (x+y) and (z+w)'s values
sw $v0, <aoff>($fp)    # store result in a
Option 2

• Eliminate nested expressions!
  • Avoids the need to push every time we have a nested expression.

• Introduce new variables to hold intermediate results
  • E.g., \( a := (x + y) + (z + w) \) might be translated to:
    
    \[
    \begin{align*}
    t0 & := x + y; \\
    t1 & := z + w; \\
    a & := t0 + t1;
    \end{align*}
    \]

• Treat temps the same as local variables
  • i.e., allocate space for temps once in the prologue and deallocate the space once in the epilogue
Option 2 example: 20 instructions
(9 memory)

\[ a := (x + y) + (z + w) \]

\[
\begin{align*}
t0 &= x + y; \\
&\quad \text{lw } \$v0, <xoff>(\$fp) \\
&\quad \text{lw } \$v1, <yoff>(\$fp) \\
&\quad \text{add } \$v0, \$v0, \$v1 \\
&\quad \text{sw } \$v0, <t0off>(\$fp)
\end{align*}
\]

\[
\begin{align*}
t1 &= z + w; \\
&\quad \text{lw } \$v0, <zoff>(\$fp) \\
&\quad \text{lw } \$v1, <woff>(\$fp) \\
&\quad \text{add } \$v0, \$v0, \$v1 \\
&\quad \text{sw } \$v0, <t1off>(\$fp)
\end{align*}
\]

\[
\begin{align*}
a &= t0 + t1; \\
&\quad \text{lw } \$v0, <t0off>(\$fp) \\
&\quad \text{lw } \$v1, <t1off>(\$fp) \\
&\quad \text{add } \$v0, \$v0, \$v1 \\
&\quad \text{sw } \$v0, <aoff>(\$fp)
\end{align*}
\]
Option 2.5

- Still doing lots of loads and stores for temps
  - (and also for variables!)
- So another idea: use registers to hold intermediate values instead of variables
  - For now:
    - Assume an infinite number of registers
    - Keep a distinction between temps and variables: variables require loading/storing, but temps do not
Option 2.5 Example

\[ a := (x + y) + (z + w) \]

t0 := x;  # load variable
t1 := y;  # load variable
t2 := t0 + t1;  # add
t3 := z;  # load variable
t4 := w;  # load variable
t5 := t3 + t4;  # add
t6 := t2 + t5;  # add
a := t6;  # store result
Option 2.5 Example: 8 instructions
(5 memory)

\[a := (x + y) + (z + w)\]

\[
\begin{align*}
t0 &:= x; \\
t1 &:= y; \\
t2 &:= t0 + t1; \\
t3 &:= z; \\
t4 &:= w; \\
t5 &:= t3 + t4; \\
t6 &:= t2 + t5; \\
a &:= t6;
\end{align*}
\]

- lw $t0,<xoff>($fp)
- lw $t1,<yoff>($fp)
- add $t2,$t0,$t1
- lw $t3,<zoff>($fp)
- lw $t4,<woff>($fp)
- add $t5,$t3,$t4
- add $t6,$t2,$t5
- sw $t6,<aoff>($fp)

- Note that each little statement translates directly to a single MIPS instruction.
Using Temps

\[ a := (x + y) + (z + w) \]

\[
\begin{align*}
t0 & := x; & \# \text{t0 taken} \\
t1 & := y; & \# \text{t0, t1 taken} \\
t2 & := t0 + t1; & \# \text{t2 taken} \\
t3 & := z; & \# \text{t2, t3 taken} \\
t4 & := w; & \# \text{t2, t3, t4 taken} \\
t5 & := t3 + t4; & \# \text{t2, t5 taken} \\
t6 & := t2 + t5; & \# \text{t6 taken} \\
a & := t6; & \# \text{<none taken>} \\
\end{align*}
\]

• We can reuse temps
• They form a stack discipline!
• Idea: Use a compile-time stack of registers instead of a run-time stack!
Using Temps

\[ a := (x + y) + (z + w) \]

\[
t0 := x; \quad \# t0 taken \\
t1 := y; \quad \# t0, t1 taken \\
t0 := t0 + t1; \quad \# t0 taken \\
t1 := z; \quad \# t0, t1 taken \\
t2 := w; \quad \# t0, t1, t2 taken \\
t1 := t1 + t2; \quad \# t0, t1 taken \\
t0 := t0 + t1; \quad \# t0 taken \\
a := t0; \quad \# <empty>
\]
Option 3

• When the compile-time stack **overflows** (i.e., need more temps than we have registers):
  • Generate code to “spill” (push) all of the temps.
  • Reset the compile-time stack to <empty>
• When the compile-time stack underflows (i.e., we need temps that we spilled earlier):
  • Generate code to pop all of the temps.
  • Reset the compile-time stack to full.
• What's really happening is that we're caching the “hot” end of the run-time stack in registers
• Some architectures (e.g., SPARC, Itanium) can do the spilling/restoring with 1 instruction.
Option 3

- Compared to previous options
  - Don’t push and pop on stack when expressions are small
  - Eliminates lots of memory access (and amortizes the cost of stack adjustment)
- But still far from optimal...
  - Consider $a + (b + (c + (d + \ldots + (y + z) \ldots) \ldots))$ versus $(\ldots((a + b) + c) + d) + \ldots + y) + z$
  - If order of evaluation doesn't matter, then want to pick one that minimizes depth of stack (i.e., less likely to overflow.)