The MIPS R4000, part 14: Common patterns

devblogs.microsoft.com/oldnewthing/20180419-00

April 19, 2018

Raymond Chen

Okay, now that we see how function calls work, we can demonstrate some common code sequences. If you are debugging through MIPS code, you'll need to be able to recognize these different types of calling sequences in order to keep your bearings.

Non-virtual calls generally look like this:

```
; Put the parameters in a0 through a3,
; and additional parameters go on the stack
; after the home space.
sw      t0, 20(sp)  ; parameter 5 passed on the stack
move    a3, s1      ; parameter 4 copied from another register
addiu   a2, sp, 32  ; parameter 3 is address of local variable
addiu   a1, t1, 1   ; parameter 2 is calculated in place
jal     destination ; call the function
move    a0, s1      ; parameter 1 copied from another register
```

The parameters could be set up in any order, and there's a good chance you'll find one of the parameters being set up in the branch delay slot. Note also that the `JAL` instruction might end up jumping to an import stub if this turns out to have been a naïvely-imported function.

Virtual calls load the destination from the target's vtable:

```
; "this" passed in a0. Other parameters go
; into a1 through a3, with additional parameters
; on the stack after the home space.
sw      t0, 20(sp)  ; parameter 5 passed on the stack
move    a3, s1      ; parameter 4 copied from another register
addiu   a2, sp, 32  ; parameter 3 is address of local variable
lw      t6, 0(a0)   ; t6 -> vtable of target
lw      t7, n(t6)   ; t7 = function pointer from vtable
jalr    t7          ; call the function
addiu   a1, t1, 1   ; parameter 2 is calculated in place
```

I put all of the virtual dispatch code in one block of contiguous instructions, but in practice the compiler may choose to interleave it with the preparation of the function arguments to avoid data load stalls. The above example uses *t6* and *t7* as temporary registers for preparing the call, but in practice, the compiler will use any volatile register that is not being used to pass parameters.

Calls to imported functions indirect through the entry in the import address table.

```
; Put the parameters in a0 through a3,
; and additional parameters go on the stack
; after the home space.
sw      t0, 20(sp)  ; parameter 5 passed on the stack
move    a3, s1      ; parameter 4 copied from another register
addiu   a2, sp, 32  ; parameter 3 is address of local variable
addiu   a1, t1, 1   ; parameter 2 is calculated in place
lui     t6, XXXX    ; t6 -> 64KB block containing import address table entry
lw      t6, YYYY(t6); t6 = function pointer from import address table entry
jalr    t6          ; call the function
move    a0, s1      ; parameter 1 copied from another register
```

Again, I put all of the relevant instructions together. In practice, the compiler tends to front-load the fetching of the function pointer.

The last interesting calling pattern for today is the jump table, commonly used for dense `switch` statements. Suppose we have this:

```
switch (n) {
case 1: ...; break;
case 2: ...; break;
case 3: ...; break;
case 4: ...; break;
}
```

The resulting code would look like this:

```
; jump to address based on value in v0
addiu   v0,v0,-1    ; subtract 1
sltiu   at,v0,4     ; in range of the jump table?
beqz    at,default  ; nope - go to default
sll     v0,v0,2     ; convert to byte offset
lui     at,XXXX     ; load high part of jump table address
addu    at,at,v0    ; add in the byte offset
lw      v0,YYYY(at) ; add in the low part and load jump table entry
jr      v0          ; and jump there
nop                 ; branch delay slot
```

The jump table pattern first performs a single-comparison range check by the standard trick of offseting the control value by the lowest value in the range and using an unsigned comparison against the length of the range. Asssuming the range check passes, we load the word at

```
address of start of jump table + 4 * index
```

The `lui` + `addu` + `lw` sequence is a pattern we saw earlier when we studied memory access: It’s the expansion of the pseudo-instruction

```
    lw      v0, XXXXYYYY(v0) ; load jump table entry
```

Once we load the jump target, we perform a register indirect jump to the intended target, and put a `nop` in the branch delay slot because we don’t have anything useful to put in there. (In practice, there might be something useful in there.)

Okay, now that we’ve seen some patterns, next time we’ll try to understand an entire function.

Raymond Chen

Follow