The Alpha AXP, part 5: Conditional operations and control flow

devblogs.microsoft.com/oldnewthing/20170811-00

August 11, 2017

Raymond Chen

The Alpha AXP has no flags register. Conditional operations are performed based on the current value of a general-purpose register. The conditions available on the Alpha AXP are the following:

`EQ`	if zero
`NE`	if not zero
`GE`	if signed greater than or equal to zero
`GT`	if signed greater than zero
`LE`	if signed less than or equal to zero
`LT`	if signed less than zero
`LBC`	if low bit clear (if even)
`LBS`	if low bit set (if odd)

In the discussion below, the abbreviation `cc` represents one of the above condition codes.

The conditional move instructions test a source register against a condition, and if the condition is true, the destination register receives the second source.

```
    CMOVcc  Ra, Rb/#b, Rc   ; if Ra meets condition, then Rc = Rb/#b
```

You can also generate booleans from conditions. Note that the set of conditions here is not the same as the standard set of conditions above!

```
CMPEQ   Ra, Rb/#b, Rc   ; Rc = (Ra == Rb/#b)
CMPLT   Ra, Rb/#b, Rc   ; Rc = (Ra < Rb/#b) signed comparison
CMPLE   Ra, Rb/#b, Rc   ; Rc = (Ra ≤ Rb/#b) signed comparison
CMPULT  Ra, Rb/#b, Rc   ; Rc = (Ra < Rb/#b) unsigned comparison
CMPULE  Ra, Rb/#b, Rc   ; Rc = (Ra ≤ Rb/#b) unsigned comparison
```

These comparison operators produce values of exactly 0 or 1, according to the result of the comparison, and the comparison is against the full 64-bit register value.

Conditional jump instructions provide a condition and a register, as well as a jump target.

```
Bcc     Ra, destination
```

where `cc` is one of the condition codes above. The instruction tests the specified register against the condition, and if true, control is transferred to the destination. The test is against the full 64-bit register value, and the destination is encoded as a 21-bit value, in units of instructions (4 bytes), which provides a reach of &pm;4MB.

Conditional branches backward are predicted taken. Conditional branches forward are predicted not taken.

There are two types of unconditional branches. They are functionally the same but have different consequences for the return address predictor.

```
BR      Ra, destination ; not expected to return
BSR     Ra, destination ; expected to return
```

These instructions store the address of the subsequent instruction (the return address) in the *Ra* register and then transfer to the destination. The `BR` instruction does not push the return address onto the return address predictor stack; the `BSR` instruction does.

The `BR` instruction is typically used with *zero* as the register to receive the return address, since the value is almost always thrown away. (Recall that there is a special exemption for branch instructions to the usual rule that instructions which write to *zero* can be optimized away.)

The Win32 calling convention dictates that the *ra* register holds the return address on entry to a function.

There are four indirect jump instructions which are all functionally equivalent but differ in their effect on the return address predictor.

```
JMP     Ra, (Rb), hint16    ; not expected to return
JSR     Ra, (Rb), hint16    ; expected to return
RET     Ra, (Rb), hint16    ; end of function
JSR_CO  Ra, (Rb), hint16    ; coroutine
```

The *Ra* register receives the return address, typically *zero* in the case of `JMP` and `RET`, and conventionally *ra* in the case of `JSR`. As you have probably guessed, `JMP` has no effect on the return address predictor, `JSR` pushes the return address onto the predictor stack, and `RET` pops the return address off of the predictor stack and predicts a transfer to the popped value. The weird guy is `JSR_CO` which replaces the return address at the top of the predictor stack with the new return address and predicts a transfer to the old value.

The official name of `JSR_CO` is `JSR_ COROUTINE`, but it doesn't really matter because I have never see `JSR_CO` in practice.

For the `JMP` and `JSR` instructions, the "hint" is a static prediction of the low 16 bits of the value in *Rb*.

The `RET` and `JSR_CO` instructions don't need a hint because they have their own return address predictor. However, DEC recommends that the hint for a `RET` instruction be 1 for a return from a procedure, and 0 otherwise. We'll see more about this another day.

The Microsoft compiler doesn't generate hints; it just sets the hint to zero. Profile-guided optimization didn't come to Visual C++ until after support for the Alpha AXP was dropped, but if it were still in support, I'm assuming that profile-guided optimization would have filled in the hint.

Non-virtual calls will look generally like this:

```
; Put the parameters in a0 through a5
; by whatever means appropriate.
; Excess parameters go on the stack.
; (Not shown here.)
BIS     zero, s1, a0    ; copied from another register
LDL     a1, 32(sp)      ; loaded from memory
ADDL    zero, #1, a2    ; calculated in place

BSR     ra, destination ; call the other function
; result is in the v0 register
```

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

```
    ; Put the parameters in a0 through a5
    ; by whatever means appropriate.
    ; Excess parameters go on the stack.
    ; (Not shown here.)
    ; "this" goes into a0.
    BIS     zero, s1, a0    ; copied from another register
    LDL     a1, 32(sp)      ; loaded from memory
    ADDL    zero, #1, a2    ; calculated in place

    LDL     t0, (a0)        ; load vtable
    LDL     t0, 8(t0)       ; load function from vtable
    BSR     ra, (t0)        ; call the function pointer
    ; result is in the v0 register
```

Calls to exported functions are indirect through a global variable, which means we need to get the address of that global.

```
    ; Put the parameters in a0 through a5
    ; by whatever means appropriate.
    ; Excess parameters go on the stack.
    ; (Not shown here.)
    BIS     zero, s1, a0    ; copied from another register
    LDL     a1, 32(sp)      ; loaded from memory
    ADDL    zero, #1, a2    ; calculated in place

    LDAH    t0, xxxx(zero)  ; 64KB block where global variable resides
    LDL     t0, yyyy(t0)    ; load the global variable
    BSR     ra, (t0)        ; call the function pointer
    ; result is in the v0 register
```

The above examples use the `LDL` instruction, which loads a register from memory. We'll learn more about memory access next time.

Raymond Chen

Follow