The Itanium processor, part 6: Calculating conditionals

devblogs.microsoft.com/oldnewthing/20150803-00

August 3, 2015

Raymond Chen

The Itanium does not have a flags register. A flags register creates implicit dependencies between instructions, which runs contrary to the highly parallel model the Itanium was designed for. Instead of implicitly setting a register after computations, the Itanium has explicit comparison operations that put the comparison result into dedicated predicate registers.

Here's a simple fragment that performs some operation if two registers are equal.

```
        cmp.eq p6, p7 = r32, r33 ;;
(p6)    something
```

The *cmp* instruction compares two values and sets the two specified predicate registers as follows:

- *p6* is *true* if the values satisfy the condition, or *false* if they do not satisfy the condition.
- *p7* is set to the opposite of *p6*

The comparison operation generates two results, one which holds the nominal result and one which holds the opposite. This lets you conditionalize both sides of a branch.

```
        cmp.eq p6, p7 = r32, r33 ;;
(p6)    something // executes if they are equal
(p7)    something // executes if they are not equal
```

There is also a *cmp4* instruction which compares two 32-bit values, in which case only the least-significant 32 bits participate in the comparison.

The comparands can be either two registers or an immediate and a register. The immediate is an 8-bit sign-extended value, though the final value may be interpreted as unsigned depending on the comparison type.

There are three comparison types:

type	meaning

eq	equality
lt	signed less than
ltu	unsigned less than

The first destination predicate register receives result of the test, and the second gets the opposite of the result.

These are the only comparisons you will see in disassembly, but the compiler can manufacture other types of comparisons. For example, if the compiler wants to perform a *ge* comparison, it can just do a *lt* comparison and flip the order of the two predicates.

More generally, the compiler can synthesize the other integer comparisons as follows:

imaginary opcode	**meaning**	**synthesized as**	
cmp.ne p, q = a, b	not equal	*cmp.eq q, p = a, b*	
cmp.ge p, q = a, b	signed greater than or equal	*cmp.lt q, p = a, b*	
cmp.gt p, q = a, b	signed greater than	*cmp.lt p, q = b, a*	if *a* is a register
		cmp.lt q, p = a − 1, b	if *a* is an immediate
cmp.le p, q = a, b	signed less than or equal	*cmp.lt q, p = b, a*	if *a* is a register
		cmp.lt p, q = a − 1, b	if *a* is an immediate
cmp.geu p, q = a, b	unsigned greater than or equal	*cmp.ltu q, p = a, b*	
cmp.gtu p, q = a, b	unsigned greater than	*cmp.ltu p, q = b, a*	if *a* is a register
		cmp.ltu q, p = a − 1, b	if *a* is an immediate
cmp.leu p, q = a, b	unsigned less than or equal	*cmp.ltu q, p = b, a*	if *a* is a register
		cmp.ltu p, q = a − 1, b	if *a* is an immediate

These syntheses rely on the identities

$x > y$	$\Leftrightarrow$	$y < x$	
$x \le y$	$\Leftrightarrow$	$\neg(x > y)$	
$x \le y$	$\Leftrightarrow$	$x - 1 < y$	for integers x and y, assuming no overflow
$x \ge y$	$\Leftrightarrow$	$y \le x$	

The next level of complexity is the parallel comparisons. These perform a comparison and combine the result with the values already in the destination predicates.

opcode	meaning	really
cmp.xx.or p, q = a, b	$p = p \mid\mid (a \text{ xx } b)$ $q = q \mid\mid (a \text{ xx } b)$	if $(a \text{ xx } b)$ then $p = q =$ *true*
cmp.xx.orcm p, q = a, b	$p = p \mid\mid \neg(a \text{ xx } b)$ $q = q \mid\mid \neg(a \text{ xx } b)$	if $\neg(a \text{ xx } b)$ then $p = q =$ *true*
cmp.xx.and p, q = a, b	$p = p \,\&\&\, (a \text{ xx } b)$ $q = q \,\&\&\, (a \text{ xx } b)$	if $\neg(a \text{ xx } b)$ then $p = q =$ *false*
cmp.xx.andcm p, q = a, b	$p = p \,\&\&\, \neg(a \text{ xx } b)$ $q = q \,\&\&\, \neg(a \text{ xx } b)$	if $(a \text{ xx } b)$ then $p = q =$ *false*
cmp.xx.or.andcm p, q = a, b	$p = p \mid\mid (a \text{ xx } b)$ $q = q \,\&\&\, \neg(a \text{ xx } b)$	if $(a \text{ xx } b)$ then $p =$ *true*, $q =$ *false*
cmp.xx.and.orcm p, q = a, b	$p = p \,\&\&\, (a \text{ xx } b)$ $q = q \mid\mid \neg(a \text{ xx } b)$	if $\neg(a \text{ xx } b)$ then $p =$ *false*, $q =$ *true*

The *meaning* column describes how it is convenient to think of the operations, but the *really* column describes how they actually work.

The `orcm` and `andcm` versions take the complement of the comparison, which is handy because some of the synthesized comparisons involve taking the opposite of the specified result.

These parallel comparisons get their name because they are designed to have multiple copies executed in parallel. Consequently, they are an exception to the general rule that you can write to a register only once per instruction group. If all writes to a predicate register are AND-like (i.e., `and` or `andcm`) or all writes are OR-like (i.e., `or` or `orcm`), then the writes are allowed to coexist within a single instruction group. (This is where the *actually* column comes in handy. You can see that all AND-like operations either do nothing or set the

predicate to *false*, and that all OR-like operations either do nothing or set the predicate to *true*. That's why they can run in parallel: If multiple conditions pass, they all do the same thing, so it doesn't matter which one goes first.)

Executing them in parallel lets you perform multiple tests in a single cycle. For example:

```
x =  ... calculate x ...;
y =  ... calculate y ...;
z =  ... calculate z ...;
if (x == 0 || y == 0 || z == 0) {
  something_is_zero;
} else {
  all_are_nonzero;
}
```

could be compiled as

```
        ... calculate x in r29 ...
        ... calculate y in r30 ...
        ... calculate z in r31 ...
        cmp.eq p6, p7 = +1, r0 ;; // set p6 = false, p7 = true

        cmp.eq.or.andcm p6, p7 = r29, r0 // p6 = p6 || x == 0
                                         // p7 = p7 && x != 0
        cmp.eq.or.andcm p6, p7 = r30, r0 // p6 = p6 || y == 0
                                         // p7 = p7 && y != 0
        cmp.eq.or.andcm p6, p7 = r31, r0 ;; // p6 = p6 || z == 0
                                         // p7 = p7 && z != 0

(p6)    something_is_zero
(p7)    all_are_nonzero
```

First, we calculate the values of *x*, *y* and *z*. At the same time, we prime the parallel comparison: we compare the constant +1 against register *r0*, which is the hard-coded zero register. This comparison always fails, so we set *p6* to *false* and *p7* to *true*.

Now we perform the three comparisons in parallel. We check if *r29*, *r30*, and *r31* are zero. If any of them is zero, then *p6* becomes *true* and *p7* becomes *false*. If all are nonzero, then nothing changes, so *p6* stays *false* and *p7* stays *true*.

Finally, we act on the calculated predicates.

Notice that the parallel comparison lets us calculate and combine all the parts of the test in a single cycle. In a flags-based architecture, we would have to perform a comparison, test the result, then perform another comparison, test the result, then perform the last comparison, and test the result one last time. That's a sequence of six dependent operations, which is difficult to parallelize. (And most likely consume three branch prediction slots instead of just one.)

The last wrinkle in the comparison instructions is the so-called unconditional comparison. This special instruction violates the rule that a predicated instruction has no effect if the predicate is false.

```
(qp)    cmp.xx.unc p, q = r, s
```

Even though there is a qualifying predicate, this comparison is executed unconditionally (as indicated by the `unc` suffix). The behavior of an unconditional comparison is

p = *qp* && (*r* xx *s*)
p = *qp* && ¬(*r* xx *s*)

In other words, if the qualifying predicate is *true*, then the instruction behaves as normal. But if the qualifying predicate is *false*, then the result of the comparison is considered *false* for all branches, regardless of the actual test.

This formulation is handy when you are nesting predicates. Consider:

```
x =  ... calculate x ...;
y =  ... calculate y ...;
if (x == 0) {
 x_is_zero;
} else {
 x_is_nonzero;
 if (y == 0) {
  x_is_nonzero_and_y_is_zero;
 } else {
  both_are_nonzero;
 }
}
```

This can be compiled like this:

```
        ... calculate x in r30 ...
        ... calculate y in r31 ...

        cmp.eq p6, p7 = r30, r0 ;;
(p6)    x_is_zero
(p7)    x_is_nonzero
(p7)    cmp.eq.unc p8, p9 = r31, r0 ;;
(p8)    x_is_nonzero_and_y_is_zero
(p9)    both_are_nonzero
```

After calculating *x* and *y*, we check whether *x* is zero. If it is, then we execute *x_is_zero*. If not, then we execute *x_is_nonzero*. Next, we check whether *y* is zero, and we do so via an unconditional comparison. That way, if we are in the case that *x* is zero, then both *p8* and *p9*

are set to *false*. Now we can use *p8* and *p9* to select between the final two branches. (Or if *x* is zero, neither gets selected.)

We'll see later that the unconditional comparison is also useful in register rotation.

So that's a quick tour of the Itanium conditional instructions. Next time, we'll start looking at speculation.

Raymond Chen

Follow