Predicated Execution

- Any operation can refer to a predicate register (see slide 27)

  <Pi> operation — *i* is the number of a predicate register (between 0 and 63)

- This means that the respective operation is to be committed (the results made visible) only when the respective predicate is true (the predicate register gets value 1).

- If the predicate value is known when the operation is issued, the operation is executed only if this value is *true*.

  If the predicate is not known at that moment, the operation is started; if the predicate turns out to be *false*, the operation is discarded.

- If no predicate register is mentioned, the operation is executed and committed unconditionally.

Predicated Execution (cont'd)

Predicate assignment

Pj, Pk = relation — *j* and *k* are predicate registers (between 0 and 63).

- Will set the value of predicate register *j* to *true* and that of predicate register *k* to *false* if the relation is evaluated to *true*; *j* will be set to *false* and *k* to *true* if the relation evaluates to *false*.

Predicated predicate assignment

<Pi> Pj, Pk = relation

i, j and *k* are predicate registers (between 0 and 63).

- Predicate registers *j* and *k* will be updated if and only if predicate register *i* is true.

The instruction format of the predicate assignment and the predicated predicate assignment operation is slightly different from that shown on slide 27. These operations have two and three fields for predicate registers, respectively.

Branch Predication

- *Branch predication* is a very aggressive *compilation technique* for generation of code with instruction level parallelism (code with parallel operations).

- Branch predication goes one step further than trace scheduling. In order to exploit all potential of parallelism in the machine it lets *operations from both branches of a conditional branch to be executed in parallel*.

- Branches can be very often eliminated and replaced by conditional execution.
  In order to do this hardware support is needed.

- Branch predication is based on the instructions for *predicated execution* provided by the Merced architecture.

  The idea is: let instructions from both branches go on in parallel, before the branch condition has been evaluated. The hardware (predicated execution) takes care that only those are committed which correspond to the right branch.

Branch Predication (cont'd)

Branch *predication* is not branch *prediction*:

- Branch prediction:
  guess for one branch and then go along that one; if the guess is bad, undo all the work;

- Branch predication:
  both branches are started and when the condition is known (the predicate registers are set) the right instructions are committed, the others are discarded.

There is no lost time with failed predictions.

Branch Predication (cont'd)

Example:

```
if (a && b)
        j = j + 1;
else{
        if (c)
                k = k + 1;
        else
                k = k - 1;
        m = k * 5}
i = i + 1;
```

Assumptions:

The values are stored in registers, as follows:
a: R0; *b*: R1; *j*: R2; *c*: R3; *k*: R4; *m*: R5; *i*: R6.

This sequence (for an ordinary processor) would be compiled to:

```
        BZ      R0, L1      branch if a == 0
        BZ      R1, L1      branch if b == 0
        ADI     R2, R2,#1   R2 ← R2 + 1;(integer)
        BR      L4
L1:     BZ      R3, L2      branch if c == 0
        ADI     R4, R4,#1   R4 ← R4 + 1;(integer)
        BR      L3
L2:     SBI     R4, R4,#1   R4 ← R4 - 1;(integer)
L3:     MPI     R5, R4,#5   R5 ← R4 * 5;(integer)
L4:     ADI     R6, R6,#1   R6 ← R6 + 1;(integer)
```

Branch Predication (cont'd)

Let us read it in this way:

```
if not(a == 0) and not(b == 0)                          ADI   R2, R2,#1
if not(not(a == 0) and not(b == 0)) and not(c == 0)     ADI   R4, R4,#1
if not(not(a == 0) and not(b == 0)) and not(not(c == 0))  SBI   R4, R4,#1
if not(not(a == 0) and not(b == 0))                     MPI   R5, R4,#5
                                                        ADI   R6, R6,#1
```

Branch Predication (cont'd)

The same with predicated execution:
(all predicates are initialised as *false*)

```
(1)  P1, P2 = EQ(R0, #0)
(2)  <P2> P1, P3 = EQ(R1, #0)
(3)  <P3> ADI R2, R2,#1
(4)  <P1> P4, P5 = NEQ(R3, #0)
(5)  <P4> ADI R4, R4,#1
(6)  <P5> SBI R4, R4,#1
(7)  <P1> MPI R5, R4,#5
(8)  ADI      R6, R6,#1
```

- The compiler can plan all these instructions to be issued in parallel, except (5) and (6) which are data-dependent.
- Instructions can be started before the particular predicate on which they depend is known. When the predicate will be known, the particular instruction will or will not be committed.

Speculative Loading

You remember when we discussed "delayed loading" (Fö. 5/6):

The load is placed so that memory latency is avoided (the value is already there when it's needed):

```
LOAD   R1,X      loads from address X into R1
ADD    R2,R1     R2 ← R2 + R1
ADD    R4,R3     R4 ← R4 + R3
SUB    R2,R4     R2 ← R2 - R4
```

```
LOAD   R1,X      loads from address X into R1
ADD    R4,R3     R4 ← R4 + R3
ADD    R2,R1     R2 ← R2 + R1
SUB    R2,R4     R2 ← R2 - R4
```