Re-write the loop code with the new instruction to remove the internal branch and the hammock. Make sure that the code always yields the same result as the original code and moreover that no unwanted exception is triggered. Explain why the new code does not have the performance problem of the original code.

d. Implementing predicated (conditional) instructions in an OoO processor such as the processor assumed in Table 3.25 is very challenging. One possible implementation is to not dispatch the CMOVZ until the value of R3 is known. However this approach has the same inefficiencies as if the branch of the hammock is not speculated, as in the original Tomasulo algorithm with no speculation. On the other hand dispatching the conditional instruction while the predicate register value is pending is fraught with trouble. Explain what the problem is and propose a solution.

Problem 3.22

This problem is about a VLIW extension of the 5-stage pipeline shown in Figure 3.48. Pipeline registers between stages are not shown but are present and are named as usual, such as ID1/EX1. Conditional branches and unconditional jumps are delayed by one long instruction and are executed in the ID4 stage in all cases, so that the long instruction following the branch in the fetch stage is always executed, whether or not the branch is taken.

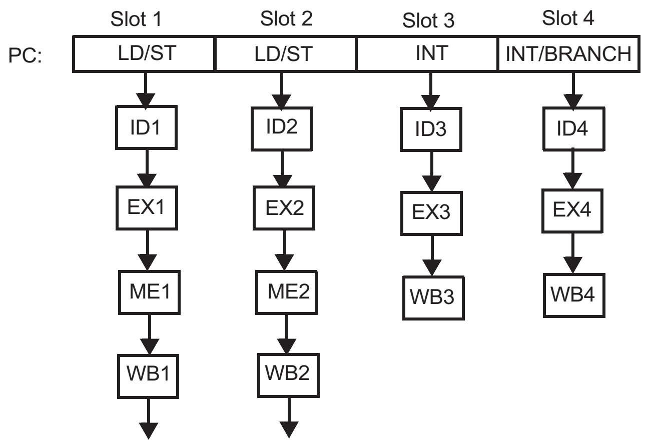

Figure 3.48. VLIW Integer pipeline

a. Compute the following operation latencies:

L.W => INT (on the register)
L.W => S.W (on the memory operand and on the address register
INT => L.W or S.W (on the address register)
INT => BRANCH (on registers)
L.W => BRANCH (on registers)

for three cases:

- No forwarding at all
- Internal register forwarding only
- Full forwarding (includes internal register forwarding)

For full forwarding, indicate all forwarding path. Forwarding is from pipeline registers to inputs of EX stages only. This is very important for the operation latencies of instruction feeding to branches

Copyright © 2011 Michel Dubois, Murali Annavaram and Per Stenström

in the case of full forwarding.

b. Consider again the code counting matches to a key in a vector of values:

```
SEARCH:     LW R5,0(R3)         /I1  Load item
            SUB R6,R5,R2        /I2  Compare with key
            BNEZ R6,NOMATCH     /I3  Check for match
            ADDI R1,R1,#1       /I4  Count matches
NOMATCH:    ADDI R3,R3,#4       /I5  Next item
            BNE R4,R3,SEARCH    /I6  Continue until all items
```

Because the branch inside the loop is an impediment to parallelization of the code, we replace it with a conditional move instruction CMOVZ:

```
SEARCH:     LW R5,0(R3)         /I1  Load item
            SUB R6,R5,R2        /I2  Compare with key
            ADDI R7,R1,#1       /I3  Assume a match
            CMOVZ R1,R7,R6      /I4  If match, then R1 is increased
            ADDI R3,R3,#4       /I5  Next item
            BNE R4,R3,SEARCH    /I6  Continue until all items
```

To further enhance parallelism the compiler unroll the loop three times. Show the best possible VLIW code for the three forwarding options. Use the same format as in Table 3.19.

c. Based on the schedule for three unrolls, can you give a formula for the execution time of the original loop body on each forwarding option up to the limit imposed by register number.

Problem 3.23

The sum prefix takes a vector X=(x_1, x_2, x_2, ..., x_n) and computes a vector Y=(y_1, y_2, y_2, ..., y_n) such that:

$$y_i = \sum_{j=1}^{i} x_j$$

for i = 1, .., n. Here is a very simple loop to compute the sum prefix in place.

```
            L.D F0,0(R1)
LOOP:       L.D F2,-8(R1)
            ADD.D F0,F2,F0
            S.D F0,-8(R1)/      O3
            SUBI R1,R1,#8
            BNEZ R1,R2 LOOP
```

R2's value is the last value of R1 minus 8. Use the same architecture as in Figure 3.27 and Table 3.17. The branch is always (correctly) predicted taken. The L1 data cache always hits.

a. At first we implement a very conservative policy in the load/store queue: a Load does not issue to cache until all possible hazards with previous Stores have been cleared. If a Load depends on a previous Store, the Load waits until the store has updated the cache (no Store-to-Load forwarding).

b. Second, assume the same conservative policy, but now with Store-to-Load forwarding: Store values are forwarded to Loads as soon as they are ready.

Copyright © 2011 Michel Dubois, Murali Annavaram and Per Stenström