UNIT - 3

INSTRUCTION –LEVEL PARALLELISM – 1: ILP

Concepts and challenges

Basic Compiler Techniques for exposing ILP

Reducing Branch costs with prediction

Overcoming Data hazards with Dynamic scheduling

Hardware-based speculation.

7 Hours

UNIT III

Instruction Level Parallelism

The potential overlap among instruction execution is called Instruction Level Parallelism (ILP) since instructions can be executed in parallel. There are mainly two approaches to exploit ILP.

i) Hardware based approach: An approach that relies on hardware to help discover and exploit the parallelism dynamically. Intel Pentium series which has dominated in the market) uses this approach.

ii) Software based approach: An approach that relies on software technology to find parallelism statically at compile time. This approach has limited use in scientific or application specific environment. Static approach of exploiting ILP is found in Intel Itanium.

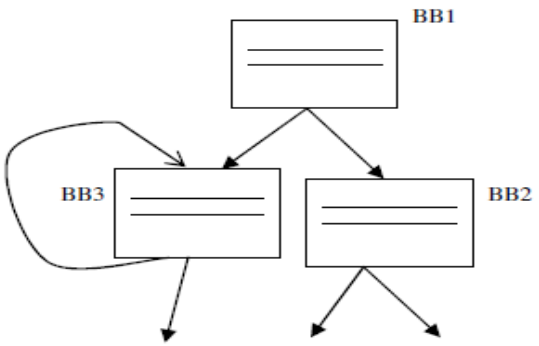
Factors of both programs and processors limit the amount of parallelism that can be exploited among instructions and these limit the performance achievable. The performance of the pipelined processors is given by:

Pipeline CPI= Ideal Pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

By reducing each of the terms on the right hand side, it is possible to minimize the overall pipeline CPI.

To exploit the ILP, the primary focus is on Basic Block (BB). The BB is a straight line code sequence with no branches in except the entry and no branches out except at the exit. The average size of the BB is very small i.e., about 4 to 6 instructions. The flow diagram segment of a program is shown below (Figure 3.1). BB1 , BB2 and BB3 are the Basic Blocks.

Figure 3.1 Flow diagram segment



The amount of overlap that can be exploited within a Basic Block is likely to be less than the average size of BB. To further enhance ILP, it is possible to look at ILP across multiple BB. The simplest and most common way to increase the ILP is to exploit the parallelism among iterations of a loop (Loop level parallelism). Each iteration of a loop can overlap with any other iteration.

Data Dependency and Hazard

If two instructions are parallel, they can execute simultaneously in a pipeline of arbitrary length without causing any stalls, assuming the pipeline has sufficient resources. If two instructions are dependent, they are not parallel and must be executed in sequential order.
There are three different types dependences.

- Data Dependences (True Data Dependency)
- Name Dependences
- Control Dependences

Data Dependences

An instruction j is data dependant on instruction i if either of the following holds:
i) Instruction i produces a result that may be used by instruction j

Eg1: i: L.D **F0**, 0(R1)
j: ADD.D F4, **F0**, F2

ith instruction is loading the data into the F0 and jth instruction use F0 as one the operand. Hence, jth instruction is data dependant on ith instruction.

Eg2: DADD **R1**, R2, R3
DSUB R4, **R1**, R5

ii) Instruction j is data dependant on instruction k and instruction k data dependant on instruction i

Eg: L.D **F4**, 0(R1)
MUL.D **F0**, **F4**, F6
ADD.D F5, **F0**, F7

Dependences are the property of the programs. A Data value may flow between instructions either through registers or through memory locations. Detecting the data flow and dependence that occurs through registers is quite straight forward. Dependences that flow through the memory locations are more difficult to detect. A data dependence convey three things.

a) The possibility of the Hazard.
b) The order in which results must be calculated and
c) An upper bound on how much parallelism can possibly exploited.

Name Dependences

A Name Dependence occurs when two instructions use the same Register or Memory location, but there is no flow of data between the instructions associated with that name.

Two types of Name dependences:

i) **Antidependence**: between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. he original ordering must be preserved to ensure that i reads the correct value.

Eg: L.D F0, 0(R1)
DADDUI R1, R1, R3

ii) **Output dependence:** Output Dependence occurs when instructions i and j write to the same register or memory location.

Ex: ADD.D **F4**, F0, F2
SUB.D **F4**, F3, F5

The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.The above instruction can be reordered or can be executed simultaneously if the name of the register is changed. The renaming can be easily done either statically by a compiler or dynamically by the hardware.

Data hazard: Hazards are named by the ordering in the program that must be preserved by the pipeline

RAW (Read After Write): j tries to read a source before i writes it, so j in correctly gets old value, this hazard is due to true data dependence.

WAW (Write After Write): j tries to write an operand before it is written by i. WAW hazard arises from output dependence.

WAR (Write After Read): j tries to write a destination before it is read by i, so that I incorrectly gets the new value. WAR hazard arises from an antidependence and normally cannot occur in static issue pipeline.

CONTROL DEPENDENCE:
A control dependence determines the ordering of an instruction i with respect to a branch instruction,

```
Ex: if P1 {
        S1;
        }
        if P2 {
        S2;
        }
```

S1 is Control dependent on P1 and

S2 is control dependent on P2 but not on P1.
a)An instruction that is control dependent on a branch cannot be moved before the branch ,so that its execution is no longer controlled by the branch.
b)An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.

BASIC PIPELINE SCHEDULE AND LOOP UNROLLING

To keep a pipe line full, parallelism among instructions must be exploited by finding sequence of unrelated instructions that can be overlapped in the pipeline. To avoid a pipeline stall,a dependent instruction must be separated from the source instruction by the distance in clock cycles equal to the pipeline latency of that source instruction. A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline.

The compiler can increase the amount of available ILP by transferring loops.

```
for(i=1000; i>0 ;i=i-1)
X[i] = X[i] + s;
```

We see that this loop is parallel by the noticing that body of the each iteration is independent.

The first step is to translate the above segment to MIPS assembly language

```
Loop: L.D F0, 0(R1) : F0=array element
ADD.D F4, F0, F2 : add scalar in F2
S.D F4, 0(R1) : store result
DADDUI R1, R1, #-8 : decrement pointer
: 8 Bytes (per DW)
BNE R1, R2, Loop : branch R1! = R2
```

Without any Scheduling the loop will execute as follows and takes 9 cycles for each iteration.

```
1 Loop: L.D F0, 0(R1) ;F0=vector element
2 stall
3 ADD.D F4, F0, F2 ;add scalar in F2
4 stall
5 stall
6 S.D F4, 0(R1) ;store result
7 DADDUI R1, R1,# -8 ;decrement pointer 8B (DW)
8 stall ;assumes can't forward to branch
        9 BNEZ R1, Loop ;branch R1!=zero
```

We can schedule the loop to obtain only two stalls and reduce the time to 7 cycles:

```
L.D F0, 0(R1)

DADDUI R1, R1, #-8
```

ADD.D F4, F0, F2

Stall

Stall

S.D F4, 0(R1)

BNE R1, R2, Loop

Loop Unrolling can be used to minimize the number of stalls. Unrolling the body of the loop by our times, the execution of four iteration can be done in 27 clock cycles or 6.75 clock cycles per iteration.

1 Loop: L.D F0,0(R1)

3 ADD.D F4,F0,F2

6 S.D 0(R1),F4 ;drop DSUBUI & BNEZ

7 L.D F6,-8(R1)

9 ADD.D F8,F6,F2

12 S.D -8(R1),F8 ;drop DSUBUI & BNEZ

13 L.D F10,-16(R1)

15 ADD.D F12,F10,F2

18 S.D -16(R1),F12 ;drop DSUBUI & BNEZ

19 L.D F14,-24(R1)

21 ADD.D F16,F14,F2

24 S.D -24(R1),F16

25 DADDUI R1,R1,#-32 :alter to 4*8

26 BNEZ R1,LOOP

Unrolled loop that minimizes the stalls to 14 clock cycles for four iterations is given below:

1 Loop: L.D F0, 0(R1)

2 L.D F6, -8(R1)

3 L.D F10, -16(R1)

4 L.D F14, -24(R1)
5 ADD.D F4, F0, F2

6 ADD.D F8, F6, F2

7 ADD.D F12, F10, F2

8 ADD.D F16, F14, F2

9 S.D 0(R1), F4

10 S.D -8(R1), F8

11 S.D -16(R1), F12

12 DSUBUI R1, R1,#32

13 S.D 8(R1), F16 ;8-32 = -24

14 BNEZ R1, LOOP

Summary of Loop unrolling and scheduling

The loop unrolling requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:

1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)

2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations
3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code

4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent

- Transformation requires analyzing memory addresses and finding that they do not refer to the same address

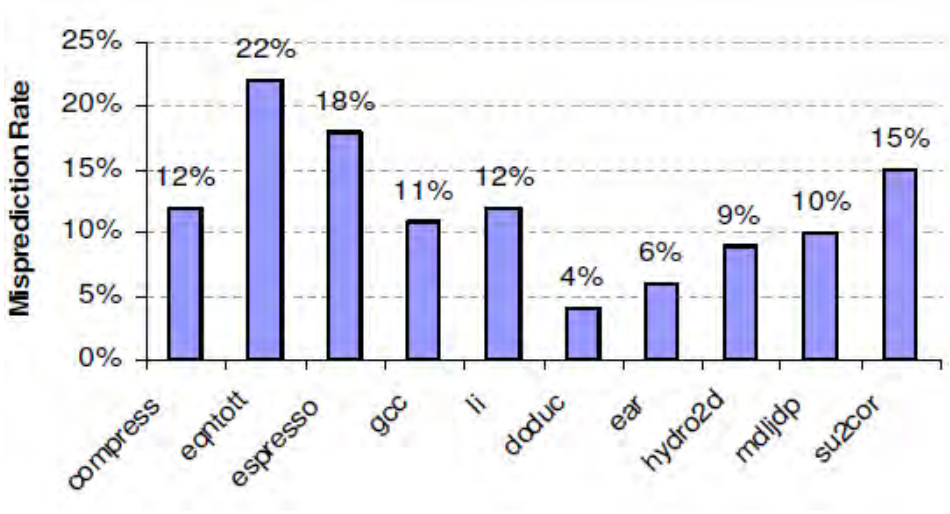
5. Schedule the code, preserving any dependences needed to yield the same result as the original code

To reduce the Branch cost, prediction of the outcome of the branch may be done. The prediction may be done statically at compile time using compiler support or dynamically using hardware support. Schemes to reduce the impact of control hazard are discussed below:

Static Branch Prediction

Assume that the branch will not be ***taken*** and continue execution down the sequential instruction stream. If the branch is ***taken***, the instruction that are being fetched and decoded must be discarded. Execution continues at the branch target. Discarding instructions means we must be able to flush instructions in the IF, ID and EXE stages.
Alternately, it is possible that the branch can be predicted as taken. As soon as the instruction decoded is found as branch, at the earliest, start fetching the instruction from the target address.

- **Average misprediction = untaken branch frequency = 34% for SPEC pgms.**



The graph shows the misprediction rate for set of SPEC benchmark programs

Dynamic Branch Prediction

With deeper pipelines the branch penalty increases when measured in clock cycles. Similarly, with multiple issue, the branch penalty increases in terms of instructions lost. Hence, a simple static prediction scheme is inefficient or may not be efficient in most of the situations. One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and if so, to begin fetching new instruction from the target address.

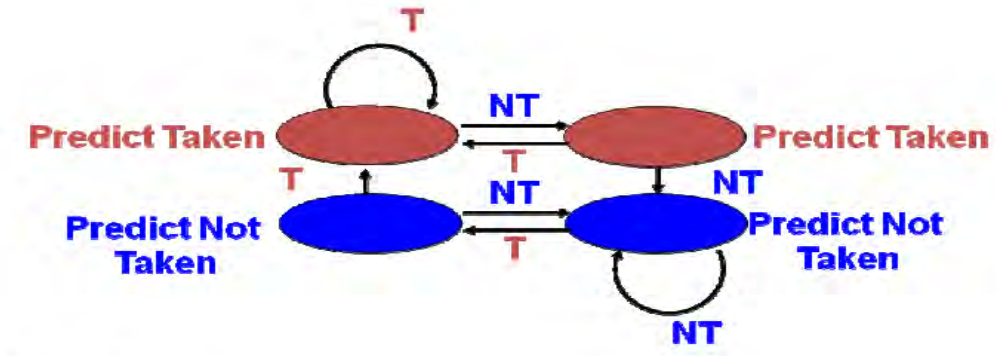
This technique is called ***Dynamic branch prediction***.
• Why does prediction work?

– Underlying algorithm has regularities
– Data that is being operated on has regularities
– Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems.
– There are a small number of important branches in programs which have dynamic behavior for which dynamic branch prediction performance will be definitely better compared to static branch prediction.

• Performance = *f*(accuracy, cost of misprediction)
• Branch History Table (BHT) is used to dynamically predict the outcome of the current branch instruction. Lower bits of PC address index table of 1-bit values
– Says whether or not branch taken last time
- No address check

• Problem: in a loop, 1-bit BHT will cause two mispredictions (average is 9 iterations before exit):
– End of loop case, when it exits instead of looping as before
– First time through loop on *next* time through code, when it predicts exit instead of looping

• Simple two bit history table will give better performance. The four different states of 2 bit predictor is shown in the state transition diagram.



The states in a 2-bit prediction scheme

Correlating Branch Predictor

It may be possible to improve the prediction accuracy by considering the recent behavior of other branches rather than just the branch under consideration. Correlating predictors are two-level predictors. Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch.

• Idea: record *m* most recently executed branches as taken or not taken, and use that pattern to select the proper *n*-bit branch history table (BHT)

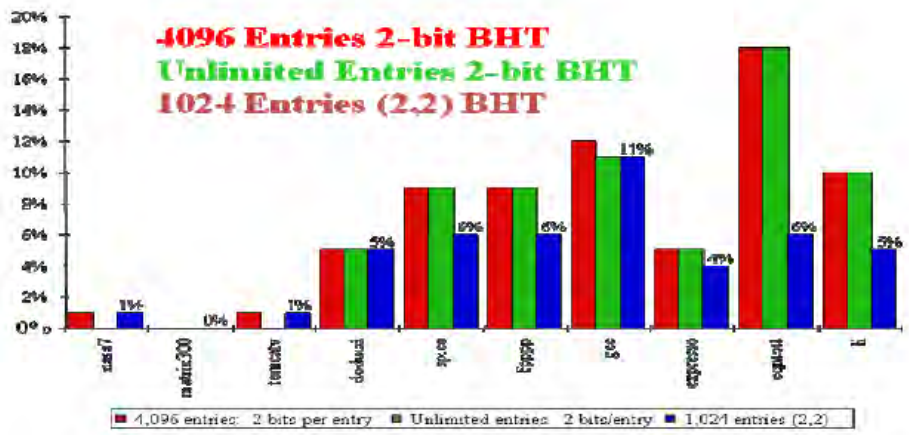
• In general, (*m*,*n*) predictor means record last *m* branches to select between 2m history tables, each with *n*-bit counters

– Thus, old 2-bit BHT is a (0,2) predictor

– Global Branch History: *m*-bit shift register keeping T/NT status of last *m* branches.

• Each entry in table has *m* *n*-bit predictors. In case of (2,2) predictor, behavior of recent branches selects between four predictions of next branch, updating just that prediction. The scheme of the table is shown:

Comparisons of different schemes are shown in the graph.



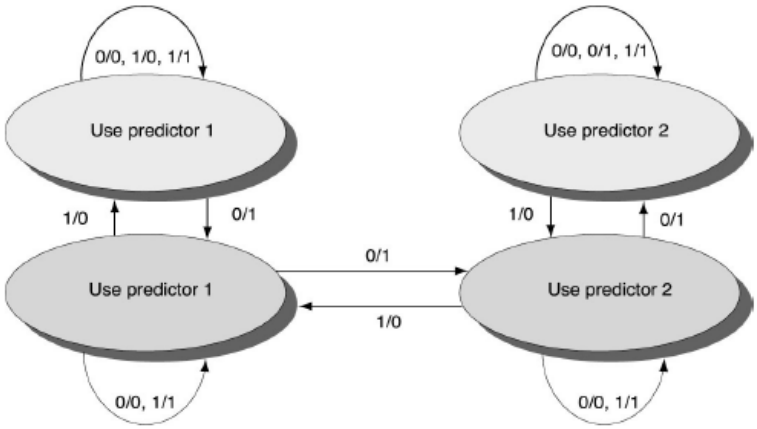
Comparison of 2 bit predictors (y-axis: % frequency of mispredictions, x-axis: SPEC89 programs)

Tournament predictor is a multi level branch predictor and uses n bit saturating counter to chose between predictors. The predictors used are global predictor and local predictor.

– Advantage of tournament predictor is ability to select the right predictor for a particular branch which is particularly crucial for integer benchmarks.

– A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks

– Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor was most effective oin recent prediction.



Dynamic Branch Prediction Summary

• Prediction is becoming important part of execution as it improves the performance of the pipeline.

• Branch History Table: 2 bits for loop accuracy

• Correlation: Recently executed branches correlated with next branch
  – Either different branches (GA)
  – Or different executions of same branches (PA)
• Tournament predictors take insight to next level, by using multiple predictors
  – usually one based on global information and one based on local information, and combining them with a selector
  – In 2006, tournament predictors using » 30K bits are in processors like the Power and Pentium 4

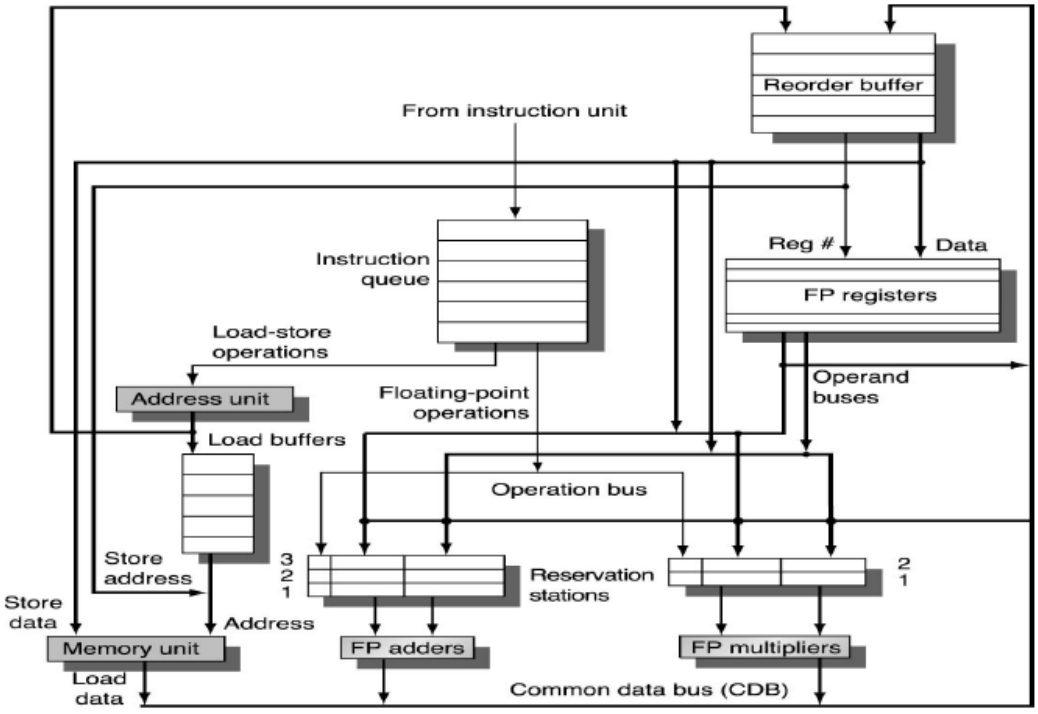
Tomasulu algorithm and Reorder Buffer

Tomasulu idea:

1. Have reservation stations where register renaming is possible
2. Results are directly forwarded to the reservation station along with the final registers. This is also called short circuiting or bypassing.

ROB:

1.The instructions are stored sequentially but we have indicators to say if it is speculative or completed execution.
2. If completed execution and not speculative and reached head of the queue then we commit it.



Speculating on Branch Outcomes

• To optimally exploit ILP (instruction-level parallelism) – e.g. with pipelining, Tomasulo,etc. – it is critical to efficiently maintain *control dependencies* (=branch dependencies)

• Key idea: *Speculate* on the outcome of branches(=predict) and execute instructions *as if* the predictions are correct

• of course, we must proceed in such a manner as to be able to recover if our speculation turns out wrong

Three components of hardware-based speculation

1. *dynamic branch prediction* to pick branch outcome
2. *speculation* to allow instructions to execute before control dependencies are resolved, i.e., before branch outcomes become known – with ability to undo in case of incorrect speculation
3. *dynamic scheduling*

Speculating with Tomasulo
Key ideas:

1. separate execution from completion: instructions to execute speculatively but no *instructions update registers or memory until no more speculative*

2. therefore, add a final step – after an instruction is no longer speculative, called *instruction commit*– when it is allowed to make register and memory updates

3. allow instructions to execute and complete out of order but force them to commit in order

4. Add hardware called the *reorder buffer* (*ROB*), with registers to hold the result of an instruction *between completion and commit*

Tomasulo's Algorithm with Speculation: Four Stages

1. Issue: get instruction from Instruction Queue
   _ if reservation station and ROB slot free (no structural hazard),
   control issues instruction to reservation station and ROB, and sends to reservation station operand values (or reservation station source for values) as well as allocated ROB slot number

2. Execution: operate on operands (EX)
_ when both operands ready then execute;if not ready, watch CDB for result

3. Write result: finish execution (WB)
_ write on CDB to all awaiting units and ROB; mark reservation station available

4. Commit: update register or memory with ROB result
   _ when instruction reaches head of ROB and results present, update register with result or store to memory and remove instruction from ROB
   _ if an incorrectly predicted branch reaches the head of ROB, flush the ROB, and restart at correct successor of branch

ROB Data Structure

ROB entry fields
• Instruction type: branch, store, register operation (i.e., ALU or load)
• State: indicates if instruction has completed and value is ready
• Destination: where result is to be written – register number for register operation (i.e. ALU or load), memory address for store
• branch has no destination result
Value: holds the value of instruction result till time to commit
Additional reservation station field
• Destination: Corresponding ROB entry number
   Example
   1. L.D F6, 34(R2)

2. L.D F2, 45(R3

3. MUL.D F0, F2, F4

4. SUB.D F8, F2, F6

5. DIV.D F10, F0, F6

6. ADD.D F6, F8, F2

The position of Reservation stations, ROB and FP registers are indicated below:

Assume latencies load 1 clock, add 2 clocks, multiply 10 clocks, divide 40 clocks
***Show data structures just before MUL.D goes to commit**...*

Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	yes	MUL	Mem[45+Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV		Mem[34+Regs[R2]]	#3		#5	

At the time MUL.D is ready to commit only the two L.D instructions have already committed,though others have completed execution
Actually, the MUL.D is at the head of the ROB – the L.D instructions are shown only for understanding purposes #X represents value field of ROB entry number X

Floating point registers

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder#	3						6		4	5
Busy	yes	no	no	no	no	no	yes	...	yes	yes

Reorder Buffer

Entry	Busy	Instruction	State	Destination	Value
1	no	L.D F6, 34(R2)	Commit	F6	Mem[34+Regs[R2]]
2	no	L.D F2, 45(R3)	Commit	F2	Mem[45+Regs[R3]]
3	yes	MUL.D F0, F2, F4	Write result	F0	#2 × Regs[F4]
4	yes	SUB.D F8, F6, F2	Write result	F8	#1 – #2
5	yes	DIV.D F10, F0, F6	Execute	F10	
6	yes	ADD.D F6, F8, F2	Write result	F6	#4 + #2

Example

```
Loop:  LD      F0      0       R1
       MULTD   F4      F0      F2
       SD      F4      0       R1
       SUBI    R1      R1      #8
       BNEZ    R1      Loop
```

Assume instructions in the loop have been issued twice
Assume L.D and MUL.D from the first iteration have committed and all other instructions have completed
Assume effective address for store is computed prior to its issue
Show data structures

Reorder Buffer

Entry	Busy	Instruction		State	Destination	Value
1	no	L.D	F0, 0(R1)	Commit	F0	Mem[0+Regs[R1]]
2	no	MUL.D	F4, F0, F2	Commit	F4	#1 × Regs[F2]
3	yes	S.D	F4, 0(R1)	Write result	0 + Regs[R1]	#2
4	yes	DADDUI	R1, R1, #-8	Write result	R1	Regs[R1] – 8
5	yes	BNE	R1, R2, Loop	Write result		
6	yes	L.D	F0, 0(R1)	Write result	F0	Mem[#4]
7	yes	MUL.D	F4, F0, F2	Write result	F4	#6 × Regs[F2]
8	yes	S.D	F4, 0(R1)	Write result	0 + #4	#7
9	yes	DADDUI	R1, R1, #-8	Write result	R1	#4 – 8
10	yes	BNE	R1, R2, Loop	Write result		

Notes

• If a branch is mispredicted, recovery is done by flushing the ROB of all entries that appear after the mispredicted branch

- entries before the branch are allowed to continue
- restart the fetch at the correct branch successor

• When an instruction commits or is flushed from the ROB then the corresponding slots become available for subsequent instructions

Advantages of hardware-based speculation:

- -able to disambiguate memory references;
- -better when hardware-based branch prediction is better than software-based branch
- prediction done at compile time; - maintains a completely precise exception model even for speculated instructions;
- does not require compensation or bookkeeping code;
  main disadvantage:
- complex and requires substantial hardware resources;