ECSE 425 – Topic 4

Advanced Pipelining: Instruction Level Parallelism and Its Exploitation

(Chapter 2 and Appendix G)

Slides: D. Patterson, W. Gross, V. Hayward, T. Arbel

1

Instruction Level Parallelism

- Pipelining overlaps the execution of instructions
- This potential overlap among instructions is called Instruction Level Parallelism (ILP)
- In this topic we look at techniques to increase the amount of ILP
- First, we will look at what limits ILP and how much we can actually expect to extract
- Then we will exploit the available ILP
- Two main techniques:
  - Hardware (market winner: Intel Pentium series)
  - Software (special niche markets, Intel Itanium, DSPs)

2

Recall from Pipelining Review

- **Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls**
  - **Ideal pipeline CPI: measure of the maximum performance attainable by the implementation**
  - **Structural hazards: HW cannot support this combination of instructions**
  - **Data hazards: Instruction depends on result of prior instruction still in the pipeline**
  - **Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)**

3

Technique	Reduces
Forwarding	Potential data hazard stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Dynamic scheduling	Data hazard stalls
Branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Speculation	Data and control stalls
Dynamic memory disambiguation	Data hazard stalls involving memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis and software pipelining	Ideal CPI and data hazard stalls

4

Instruction-Level Parallelism (ILP)

- Basic Block (BB) ILP is quite small
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - average dynamic branch frequency 15% to 25% => 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks
- Simplest: <u>loop-level parallelism</u> to exploit parallelism among iterations of a loop
  - Vector is one way
  - If not vector, then either dynamic via branch prediction or static via loop unrolling by compiler

5

Data Dependence and Hazards

- $Instr_J$ is data dependent on $Instr_I$
  $Instr_J$ tries to read operand before $Instr_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- or $Instr_J$ is data dependent on $Instr_K$ which is dependent on $Instr_I$
- Caused by a "True Dependence" (compiler term)
- If true dependence caused a hazard in the pipeline, called a Read After Write (RAW) hazard

6

Data Dependence and Hazards

- Dependences are a property of programs
- Presence of dependence indicates potential for a hazard, but actual hazard and length of any stall is a property of the pipeline
- Importance of the data dependencies

1) indicates the possibility of a hazard

2) determines order in which results must be calculated

3) sets an upper bound on how much parallelism can possibly be exploited

- Today looking at HW schemes to avoid hazard

7

Name Dependence #1: Anti-dependence

- Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence
- $Instr_J$ writes operand ***before*** $Instr_I$ reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1"

- If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard

8

Name Dependence #2: Output dependence

- $Instr_J$ writes operand ***before*** $Instr_I$ writes it.

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "output dependence" by compiler writers This also results from the reuse of name "r1"
- If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard

ILP and Data Hazards

- HW/SW must preserve program order: order instructions would execute in if executed sequentially 1 at a time as determined by original source program
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program
- Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict
  - Register renaming resolves name dependence for regs
  - Either by compiler or by HW

Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

```
if p1 {
  S1;
};
if p2 {
  S2;
}
```

- `S1` is control dependent on `p1`, and `S2` is control dependent on `p2` but not on `p1`.

11

Control Dependence Ignored

- Control dependence need not be preserved
  - willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program
- Instead, 2 properties critical to program correctness are exception behavior and data flow

12

Exception Behavior

- Preserving exception behavior => any changes in instruction execution order must not change how exceptions are raised in program (=> no new exceptions)
- Example:

```
    DADDU     R2,R3,R4
    BEQZ      R2,L1
    LW        R1,0(R2)
L1:
```

- Problem with moving `LW` before `BEQZ`?

13

Data Flow

- Data flow: actual flow of data values among instructions that produce results and those that consume them
  - branches make flow dynamic, determine which instruction is supplier of data
- Example:

```
DADDU    R1,R2,R3
BEQZ     R4,L
DSUBU    R1,R5,R6
L:  …
OR       R7,R1,R8
```

- `OR` depends on `DADDU` or `DSUBU`?
  Must preserve data flow on execution

14

Basic Compiler Techniques for Exposing ILP

15

Basic Compiler Scheduling

- The idea: keep the pipeline full
  - Avoid stalls due to hazards
- Scheduling
  - find a sequence of instructions that can be overlapped in the pipeline
- We will look at scheduling in the compiler. The hardware then executes the scheduled code in-order
- How do we achieve our goal of keeping the pipeline full ?

16

Basic Compiler Scheduling

- A dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of the source instruction
- For example, in a pipeline with forwarding
  - latency of the EX stage (ALU) is 0.
  - The data memory latency is 1
- A compiler's ability to perform this scheduling depends on:
  - The amount of ILP in the program
  - The latencies of the functional units

17

Basic Compiler Scheduling

- Assume the classic 5-stage integer pipeline
- Integer ALU latency is 0 CC
- Integer load latency is 1 CC
- Branch delay is 1 CC
- Fully pipelined FUs (assume no structural hazards)
- Assume the following FP latencies (averages):

Producer	Consumer	Latency (CCs)
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

18

Loop Example

- Adding a scalar to a vector (loop is parallel since the body of each iteration is independent)

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

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


Loop Example

- Ignore delayed branches
- Unscheduled code: 9 clock cycles

```
1   Loop:      L.D       F0,0(R1)
2              stall
3              ADD.D     F4,F0,F2
4              stall
5              stall
6              S.D       F4,0(R1)
7              DADDUI    R1,R1,#-8
8              stall
9              BNE       R1,R2,Loop
```

Loop Example

- Scheduled code: 7 cycles
- Not trivial: S.D. depends on DAADUI. Swap them but change address

```
1  Loop:    L.D      F0,0(R1)
2           DADDUI   R1,R1,#-8
3           ADD.D    F4,F0,F2
4           stall
5           stall
6           S.D      F4,8(R1)   ; altered
7           BNE      R1,R2,Loop ; delayed branch
```

21

Loop Example

- 1 branch delay slot
- Unscheduled code: 10 clock cycles

```
1   Loop:    L.D      F0,0(R1)
2            stall
3            ADD.D    F4,F0,F2
4            stall
5            stall
6            S.D      F4,0(R1)
7            DADDUI   R1,R1,#-8
8            stall
9            BNE      R1,R2,Loop
10           stall
```

22

Loop Example

- **Scheduled code: 6 cycles**
- **Problem: only doing work on the array element in 3/6 cycles. Other 3 are for loop overhead**

```
Loop:    L.D      F0,0(R1)
         DADDUI   R1,R1,#-8
         ADD.D    F4,F0,F2
         stall
         BNE      R1,R2,Loop ; delayed branch
         S.D      F4,8(R1)   ; altered
```


Loop Unrolling

- ***Unroll* the loop**
  - **Replicate the body of the loop many times**
  - **Adjust the loop termination code**
- **Eliminating the branch allows instructions from different iterations to be scheduled together**
  - **In this case we can eliminate the data stall**

Unroll Loop Four Times (straightforward way)

Rewrite loop to minimize stalls?

```
Loop:L.D    F0,0(R1)
     ADD.D  F4,F0,F2
     S.D    F4,0(R1)      ;drop DADDUI & BNE
     L.D    F6,-8(R1)
     ADD.D  F8,F6,F2
     S.D    F8,-8(R1)     ;drop DADDUI & BNE
     L.D    F10,-16(R1)
     ADD.D  F12,F10,F2
     S.D    F12,-16(R1    ;drop DADDUI & BNE
     L.D    F14,-24(R1)
     ADD.D  F16,F14,F2
     S.D    F16,-24(R1)
     DADDUI R1,R1,#-32    ;alter to 4*8
     BNE    R1,R2,LOOP
```

1 cycle stall

2 cycles stall

1 cycle stall

14 + 4x(1+2) + 2= 28 clock cycles, or 7 per iteration

Assumes R1 is multiple of 32 (# loops a multiple of 4)

Textbook example

- **The textbook on page 77-78 does the same example, but without branch delay**
- **27 clock cycles (6.75 cycles per iteration)**
- **Work through it to understand the difference of one clock cycle**

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
  - For large values of n, most of the execution time will be spent in the unrolled loop

Unrolled Loop That Minimizes Stalls

```
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    F4,0(R1)
10     S.D    F8,-8(R1)
11     DADDUI R1,R1,#-32
12     S.D    F12,16(R1)
13     BNE    R1,R2,LOOP
14     S.D    F16,8(R1) ; 8-32 = -24
```

- What assumptions made when moved code?
  - OK to move store past DADDUI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration (textbook without branch delay has a different schedule but also is able to do it in 14 cycles – work through it)

Compiler Perspectives on Code Movement

- **Compiler concerned about dependencies in program**
- **Whether or not a HW hazard depends on pipeline**
- **Try to schedule to avoid hazards that cause performance losses**
- **(True) Data dependencies (RAW if a hazard for HW)**
  - Instruction i produces a result used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- **If dependent, can't execute in parallel**
- **Easy to determine for registers (fixed names)**
- **Hard for memory ("memory disambiguation" problem):**
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?

Where are the name dependencies?

```
Loop:L.D    F0,0(R1)
     ADD.D  F4,F0,F2
     S.D    F4,0(R1)     ;drop DADDUI & BNE
     L.D    F0,-8(R1)
     ADD.D  F4,F0,F2
     S.D    F4,-8(R1)    ;drop DADDUI & BNE
     L.D    F0,-16(R1)
     ADD.D  F4,F0,F2
     S.D    F4,-16(R1)   ;drop DADDUI & BNE
     L.D    F0,-24(R1)
     ADD.D  F4,F0,F2
     S.D    F4,-24(R1)
     DADDUI R1,R1,#-32   ;alter to 4*8
     BNE    R1,R2,LOOP
     NOP
```

How can remove them?

Where are the name dependencies?

```
Loop:L.D    F0,0(R1)
     ADD.D  F4,F0,F2
     S.D    F4,0(R1)     ;drop DADDUI & BNE
     L.D    F6,-8(R1)
     ADD.D  F8,F6,F2
     S.D    F8,-8(R1)    ;drop DADDUI & BNE
     L.D    F10,-16(R1)
     ADD.D  F12,F10,F2
     S.D    F12,-16(R1)  ;drop DADDUI & BNE
     L.D    F14,-24(R1)
     ADD.D  F16,F14,F2
     S.D    F16,-24(R1)
     DADDUI R1,R1,#-32   ;alter to 4*8
     BNE    R1,R2,LOOP
     NOP
```

"register renaming"

Compiler Perspectives on Code Movement

- Name dependencies are hard to discover for memory Accesses
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- Our example required compiler to know that if R1 doesn't change then:

  0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)

  There were no dependencies between some loads and stores so they could be moved by each other

Steps Compiler Performed to Unroll

- Check OK to move the S.D after DADDUI and BNEZ, and find amount to adjust S.D offset
- Determine unrolling the loop would be useful by finding that the loop iterations were independent
- Rename registers to avoid name dependencies
- Eliminate extra test and branch instructions and adjust the loop termination and iteration code
- Determine loads and stores in unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
  - requires analyzing memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield same result as the original code

33

Drawbacks

- Code length (an issue for embedded processors)
- Uses lots of registers
  - "Register pressure"
  - Could be a problem with aggressive unrolling and scheduling

34

Reducing branch costs with Branch prediction

35

Branch Prediction

- **The fundamental problem:**
  - There is a delay between the cycle which we find out if the instruction is a branch, what it's target is, whether it is taken or not,…..and the cycle from which we need to fetch the next instruction.
- **One way to get around this is to *guess* whether a branch is taken or not taken…if we are correct then there could potentially be no penalty.**
- **We suffer a penalty if the guess was wrong**

36

Looking Ahead…

- To lower the IDEAL CPI, we will consider machines that can ISSUE more than one instruction in a clock cycle...
  - "multiple issue" (Superscalar and VLIW)

37

Case for Branch Prediction when Issue N instructions per clock cycle

1. Branches will arrive up to *n* times faster in an *n*-issue processor
2. Amdahl's Law => relative impact of the control stalls will be larger with the lower potential CPI in an *n*-issue processor

38

Static Branch Prediction

- We saw this idea earlier
  - Delayed branches

```
        LD      R1,0(R2)
        DSUBU   R1,R1,R3
        BEQZ    R1,L
        NOP
        OR      R4,R5,R6
        DADDU   R10,R4,R3
L:      DADDU   R7,R8,R9
```

39

Static Branch Prediction Strategies

- Predict-taken
  - Midprediction rate = untaken branch frequency
  - SPEC: 34% misprediction (9% to 59%)
- Predict based on branch direction
  - E.g. predict forward-going branches as not taken and backwards-going branches as taken
- Collect profile information by running the program a few times. Recompile with this profile information.
  - Studies have showed that even when the data changes the profile is pretty accurate

40

Static Branch Prediction

- Static branch prediction is useful when:
  1. Branch delays are exposed by architecture
  2. Assisting dynamic predictors (IA-64)
  3. Determining which code paths are more frequent (for code scheduling)

41

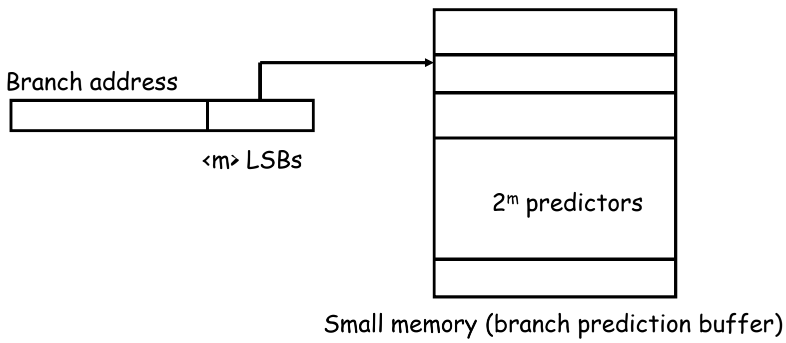
The case for dynamic branch prediction

- The performance of branch prediction rests on how accurate our predictions are.
- We have seen a compiler scheme for filling the branch delay (static branch prediction).
- Analyze each branch and try to fill the delay slot with an instruction from the branch target or the fall through.
- The problem is… it is very hard to predict the direction of branches in the compiler…we really need to consider the dynamic branch behaviour.

42

Dynamic Branch Prediction

- **IDEA: predict the outcome of a branch based on its past behaviour**



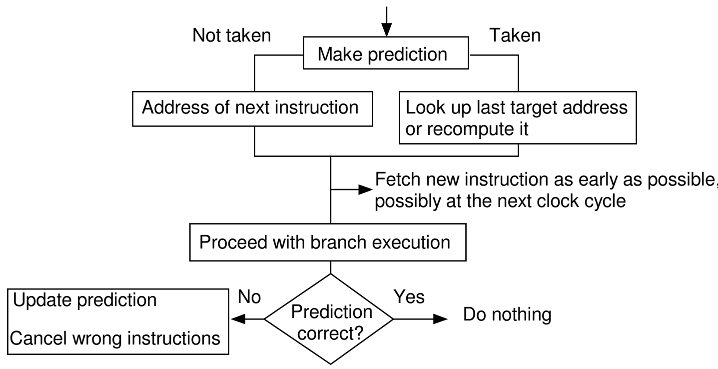
43

7 Branch Prediction Schemes

1. **1-bit Branch-Prediction Buffer**
2. **2-bit Branch-Prediction Buffer**
3. **Correlating Branch Prediction Buffer**
4. **Tournament Branch Predictor**
5. **Branch Target Buffer**
6. **Integrated Instruction Fetch Units**
7. **Return Address Predictors**

44

Dynamic Branch Prediction



Performance = *f*(accuracy, cost of misprediction)

45

1-bit Predictors

- **Branch History Table: Lower bits of PC address index table of 1-bit values**
  - Says whether or not branch taken last time
  - No address check (saves HW, but may not be right branch)
  - Adequate performance for numerical code with many loops
- **Problem: in a loop, 1-bit BHT will cause 2 mispredictions**
  - 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
  - Only 80% accuracy even if loop 90% of the time

46

Example

- Loop with 10 iterations. First 9 are taken and then the last is not.
- Mispredict 2 times for every 10 instructions
- 80% prediction accuracy
- (mispredict at twice the rate of branch not taken…should be able to at least match the taken branch frequency for highly regular loops)

…TTT N TTTTTTTTT N TTT…

mispredictions

1-bit predictors

- Prediction is wrong whenever there is a transition in the branching pattern.
- Example
  - NTNTNT
  - 1-bit predictor is never correct ! (0%)
  - Tossing a coin (no prediction at all) gives 50%
- However, real code has bias
- A branch taken several times is likely to be taken again
- Solution: keep more “memory” than is possible by just one bit…try two bits

2-bit predictors

- Count the number of 'taken' (not taken) outcomes
- Two taken (not taken) in a row → predict "taken" (not taken)
- A single not taken (taken) branch will not affect the prediction – there need to be two in a row to affect the prediction
- In general, with n prediction bits, it takes 2^{n-1} mispredictions before the predictor changes its mind

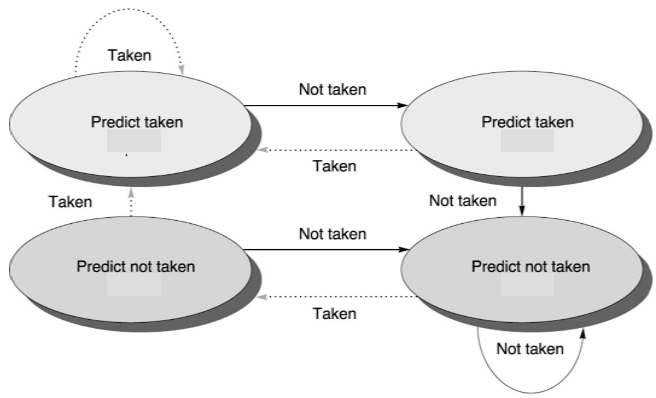
49

Examples

- …NNNNN   TNTNTN   TTTTTT…
- 50 % prediction accuracy
- …TTTN   TTTTTTTTT N   TTT…
- 90% prediction accuracy

50

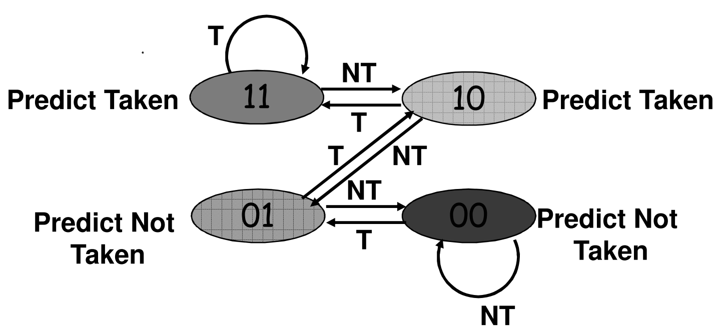
2-Bit Branch Prediction



- A branch that strongly favours taken or not taken will be mispredicted less often than with a 1-bit predictor

51

Counter Implementation



52

Accuracy of 2-bit predictors

- 99-100% on heavy matrix code
- 80 – 90% on integer code (e.g. gcc)
- Statistics show virtualy no gain in accuracy with more states and buffers of more than 1K entries.
- However, there are some cases where we can do better…

53

Correlating Branch Predictors

- Why is the performance of integer code so low?
- We assumed that different branches' behaviour was not correlated..
- But, they often are…

```
If (a == 2)
   a =0
If( b == 2)
   b= 0
If (a != b)
   …
```

- A simple predictor that considers only one branch can't capture this behaviour

54

Correlating Branch Predictors

- Idea: taken/not taken of recently executed branches is related to behavior of next branch (as well as the history of that branch behavior)
- Simple predictor: keep a history of 1 branch and each predictor is 1-bit (1,1)

55

Example without Correlation

- E.g.

```
B1:   If (d == 0)
         d = 1
B2:   If (d == 1)
         …
```

Try a 1-bit predictor

	Branch	Pred	outcome	update
d=2	B1	N	N	N
	B2	N	N	N
d=0	B1	[redacted]	[redacted]	T
	B2	[redacted]	[redacted]	T
d=2	B1	[redacted]	[redacted]	N
	B2	[redacted]	[redacted]	N
d=0	B1	[redacted]	[redacted]	T
	B2	[redacted]	[redacted]	T

56

Simple Correlating Predictor

Each branch has 2 1-bit predictors yielding four possibilities: (1,1) correlating branch predictor

Prediction bits	
Use this one if last branch **not taken**	Use this one if last branch **taken**
N	N
N	T
T	N
T	T

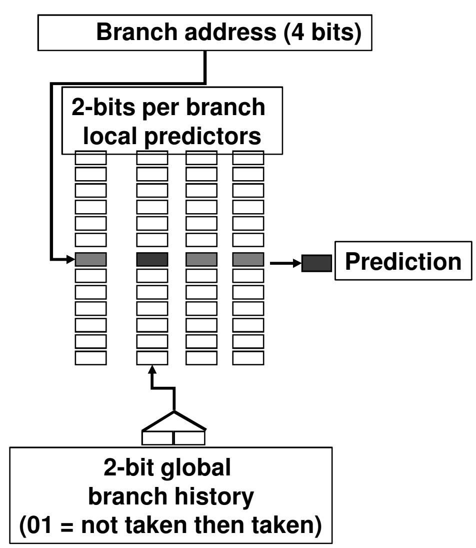
	Branch	Pred. Bits	Prediction	Outcome	Update
d=2	B1	N N	N	N	NN
	B2	N N	N	N	NN
d=0	B1	NN	N	T	TN
	B2	N N	N	T	NT
d=2	B1	TN	N	N	TN
	B2	NT	N	N	NT
d=0	B1	TN	T	T	TN
	B2	NT	T	T	NT

57

Correlating Branches

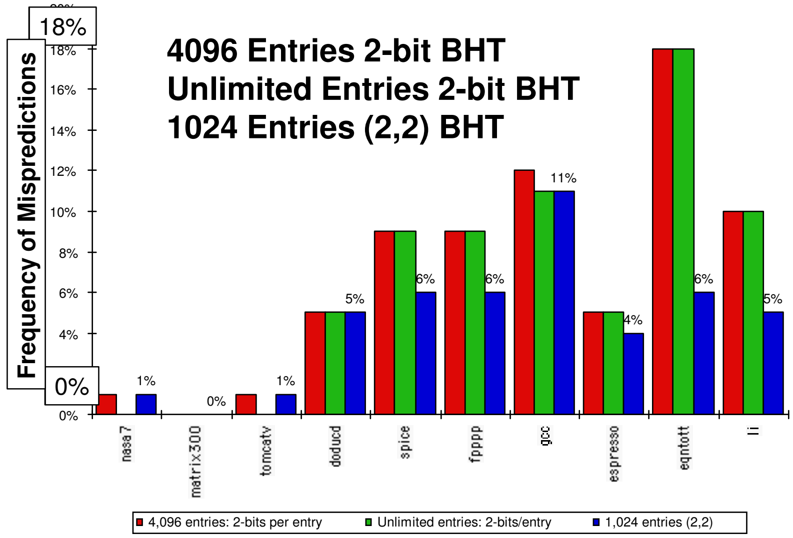
Behavior of recent branches selects between, say, 4 predictions of next branch, updating just that prediction

- (2,2) predictor: 2-bit global, 2-bit local
- General: (m,n) uses behaviour of the last m branches to choose from 2^m predictors each of which is an n-bit predictor



58

Accuracy of Different Schemes



59

BHT Accuracy

- **Mispredict because either:**
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- **4096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%**
- **For SPEC92, 4096 about as good as infinite table**

60

Tournament Predictors

- Motivation for correlating branch predictors is 2-bit predictor failed on important branches; by adding global information, performance improved
- Tournament predictors: use 2 predictors, 1 based on global information and 1 based on local information, and combine with a selector
- Hopes to select right predictor for right branch
- Pentium 4 and Power5 – 30K bits tournament predictors

61

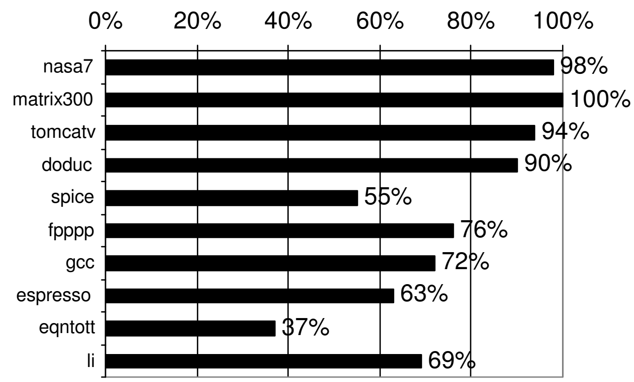
Tournament Predictor in Alpha 21264

- 4K 2-bit counters to choose from among a global predictor and a local predictor
- Global predictor also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor
  - 12-bit pattern: ith bit 0 => ith prior branch not taken;
    ith bit 1 => ith prior branch taken;
- Local predictor consists of a 2-level predictor:
  - Top level a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns 10 branches to be discovered and predicted.
  - Next level Selected entry from the local history table is used to index a table of 1K entries consisting a 3-bit saturating counters, which provide the local prediction
- Total size: 4K*2 + 4K*2 + 1K*10 + 1K*3 = 29K bits!

  (~180,000 transistors)

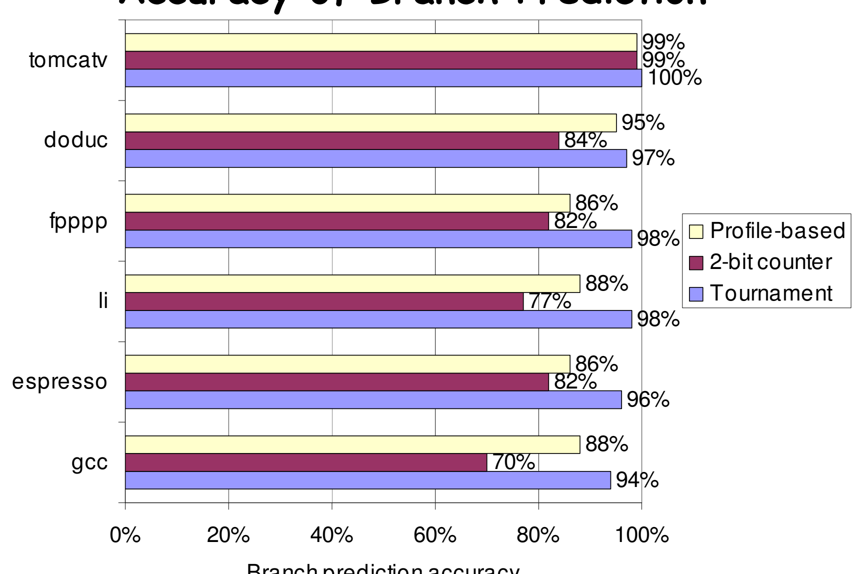
62

% of predictions from local predictor in Tournament Prediction Scheme



63

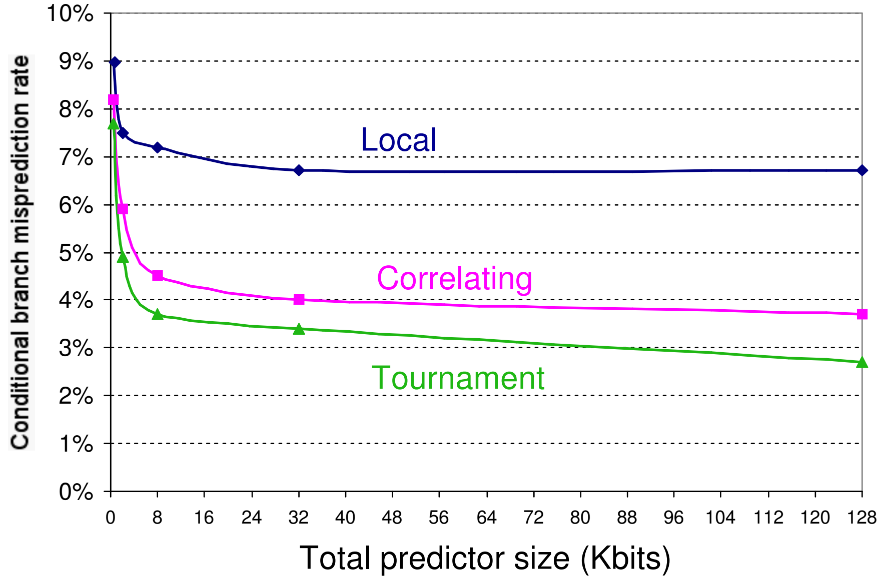
Accuracy of Branch Prediction



- **Profile: branch profile from last execution (static in that in encoded in instruction, but profile)**

64

Accuracy v. Size (SPEC89)



65

Dynamic Branch Prediction Summary

- **Prediction becoming important part of scalar execution**
- **Branch History Table: 2 bits for loop accuracy**
- **Correlation: Recently executed branches correlated with next branch.**
  - **Either different branches**
  - **Or different executions of same branches**
- **Tournament Predictor: more resources to competitive solutions and pick between them**

66

Dynamic Scheduling

67

Advantages of Dynamic Scheduling

- Handles cases when dependences unknown at compile time
  - (e.g., because they may involve a memory reference)
- It simplifies the compiler
- Allows code that compiled for one pipeline to run efficiently on a different pipeline
- Hardware speculation, a technique with significant performance advantages, that builds on dynamic scheduling

68

HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stall to proceed

```
DIVD  F0,F2,F4
ADDD  F10,F0,F8
SUBD  F12,F8,F14
```

- Enables out-of-order execution
  and allows out-of-order completion
- Will distinguish when an instruction *begins execution* and when it *completes execution*; between 2 times, the instruction is *in execution*
- In a dynamically scheduled pipeline, all instructions pass through issue stage in order (in-order issue)

69

Dynamic Scheduling Step 1

- Simple pipeline had 1 stage to check both structural and data hazards: Instruction Decode (ID), also called Instruction Issue
- Split the ID pipe stage of simple 5-stage pipeline into 2 stages:
- *Issue*—Decode instructions, check for structural hazards
- *Read operands*—Wait until no data hazards, then read operands

70

A Dynamic Algorithm: Tomasulo's Algorithm

- For IBM 360/91 (before caches!)
- Goal: High Performance without special compilers
- Small number of floating point registers (4 in 360) prevented interesting compiler scheduling of operations
  - This led Tomasulo to try to figure out how to get more effective registers — renaming in hardware!
- Why Study 1966 Computer?
- The descendants of this have flourished!
  - Alpha 21264, HP 8000, MIPS 10000, Pentium III, PowerPC 604, …

71

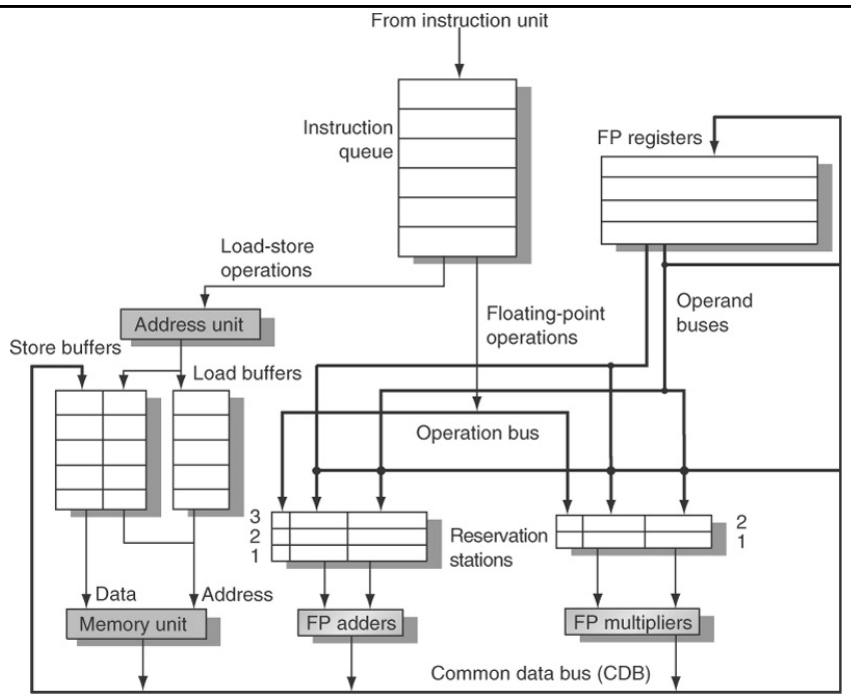
Tomasulo Algorithm

- Control & buffers distributed with Function Units (FU)
  - FU buffers called "reservation stations"; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations(RS); called register renaming ;
  - avoids WAR, WAW hazards
  - More reservation stations than registers, so can do optimizations compilers can't
- Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs
- Load and Stores treated as FUs with RSs as well
- Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue

72

Tomasulo Organization

73



Dynamic Scheduling

- Handles cases when dependences unknown at compile time
  - (e.g., because they may involve a memory reference)
- It simplifies the compiler
- Allows code that compiled for one pipeline to run efficiently on a different pipeline
- Hardware speculation, a technique with significant performance advantages, that builds on dynamic scheduling

75

Reservation Station Components

Op: Operation to perform in the unit (e.g., + or –)

Vj, Vk: Value of Source operands
- Store buffers has V field, result to be stored

Qj, Qk: Reservation stations producing source registers (value to be written)
- Note: Qj,Qk=0 => ready
- Store buffers only have Qi for RS producing result

Busy: Indicates reservation station or FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

76

Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).
2. Execute—operate on operands (EX)
   When both operands ready then execute; if not ready, watch Common Data Bus for result
3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting units; mark reservation station available

- Normal data bus: data + destination ("go to" bus)
- Common data bus: data + source ("come from" bus)
  - 64 bits of data + 4 bits of Functional Unit source address
  - Write if matches expected Functional Unit (produces result)
  - Does the broadcast
- Example speed:
  3 clocks for Fl .pt. +,-; 10 for * ; 40 clks for /

77

Tomasulo Example

Instruction stream

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2			
LD F2	45+	R3			
MULTD F0	F2	F4			
SUBD F8	F6	F2			
DIVD F10	F0	F6			
ADDD F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

3 Load/Buffers

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

FU count down

3 FP Adder R.S.
2 FP Mult R.S.

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
0	FU									

Clock cycle counter

78

Tomasulo Example Cycle 1

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1		
LD F2	45+	R3			
MULTD F0	F2	F4			
SUBD F8	F6	F2			
DIVD F10	F0	F6			
ADDD F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
1	FU				Load1					

79

Tomasulo Example Cycle 2

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1		
LD F2	45+	R3	2		
MULTD F0	F2	F4			
SUBD F8	F6	F2			
DIVD F10	F0	F6			
ADDD F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU		Load2		Load1					

Note: Can have multiple loads outstanding

80

Tomasulo Example Cycle 3

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1					

- **Note: registers names are removed ("renamed") in Reservation Stations; MULT issued**
- **Load1 completing; what is waiting for Load1?**

Tomasulo Example Cycle 4

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(A1)	Add1				

- **Load2 completing; what is waiting for Load2?**

Tomasulo Example Cycle 5

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Mult1	M(A2)		M(A1)	Add1	Mult2			

- Timer starts down for Add1, Mult1

Tomasulo Example Cycle 6

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- Issue ADDD here despite name dependency on F6?

Tomasulo Example Cycle 7

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4	7	
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- **Add1 (SUBD) completing; what is waiting for it?**

85

Tomasulo Example Cycle 8

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4	7	8
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1	M(A2)		Add2	(M-M)	Mult2			

86

Tomasulo Example Cycle 9

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	FU	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 10

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
10	FU	Mult1	M(A2)		Add2	(M-M)	Mult2			

- **Add2 (ADDD) completing; what is waiting for it?**

Tomasulo Example Cycle 11

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result			Busy	Address
LD	F6	34+	R2	1	3	4		Load1	No	
LD	F2	45+	R3	2	4	5		Load2	No	
MULTD	F0	F2	F4	3				Load3	No	
SUBD	F8	F6	F2	4	7	8				
DIVD	F10	F0	F6	5						
ADDD	F6	F8	F2	6	10	11				

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- Write result of ADDD here?
- All quick instructions complete in this cycle!

89

Tomasulo Example Cycle 12

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result			Busy	Address
LD	F6	34+	R2	1	3	4		Load1	No	
LD	F2	45+	R3	2	4	5		Load2	No	
MULTD	F0	F2	F4	3				Load3	No	
SUBD	F8	F6	F2	4	7	8				
DIVD	F10	F0	F6	5						
ADDD	F6	F8	F2	6	10	11				

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

90

Tomasulo Example Cycle 13

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result		Busy	Address
LD	F6	34+	R2	1	3	4	Load1	No	
LD	F2	45+	R3	2	4	5	Load2	No	
MULTD	F0	F2	F4	3			Load3	No	
SUBD	F8	F6	F2	4	7	8			
DIVD	F10	F0	F6	5					
ADDD	F6	F8	F2	6	10	11			

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
13	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

91

Tomasulo Example Cycle 14

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result		Busy	Address
LD	F6	34+	R2	1	3	4	Load1	No	
LD	F2	45+	R3	2	4	5	Load2	No	
MULTD	F0	F2	F4	3			Load3	No	
SUBD	F8	F6	F2	4	7	8			
DIVD	F10	F0	F6	5					
ADDD	F6	F8	F2	6	10	11			

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

92

Tomasulo Example Cycle 15

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- **Mult1 (MULTD) completing; what is waiting for it?**

93

Tomasulo Example Cycle 16

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
16	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- **Just waiting for Mult2 (DIVD) to complete**

94

Faster than light computation
(skip a couple of cycles)

95

Tomasulo Example Cycle 55

Instruction status:

Instruction	j	k	Issue	Exec Comp	Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3	15	16
SUBD F8	F6	F2	4	7	8
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
55	FU	M*F4	M(A2)		(M-M+N	(M-M)	Mult2			

96

Tomasulo Example Cycle 56

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result		Busy	Address
LD	F6	34+	R2	1	3	4	Load1	No	
LD	F2	45+	R3	2	4	5	Load2	No	
MULTD	F0	F2	F4	3	15	16	Load3	No	
SUBD	F8	F6	F2	4	7	8			
DIVD	F10	F0	F6	5	56				
ADDD	F6	F8	F2	6	10	11			

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- **Mult2 (DIVD) is completing; what is waiting for it?**

Tomasulo Example Cycle 57

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result		Busy	Address
LD	F6	34+	R2	1	3	4	Load1	No	
LD	F2	45+	R3	2	4	5	Load2	No	
MULTD	F0	F2	F4	3	15	16	Load3	No	
SUBD	F8	F6	F2	4	7	8			
DIVD	F10	F0	F6	5	56	57			
ADDD	F6	F8	F2	6	10	11			

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Result			

- **Once again: In-order issue, out-of-order execution and out-of-order completion.**

Tomasulo Drawbacks

- Complexity
  - delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620 in CA:AQA 2/e, but not in silicon!
- Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
  - Each CDB must go to multiple functional units ⇒high capacitance, high wiring density
  - Number of functional units that can complete per cycle limited to one!
    - » Multiple CDBs ⇒ more FU logic for parallel assoc stores
- Non-precise interrupts!
  - We will address this later

99

Tomasulo Loop Example

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

- This time assume Multiply takes 4 clocks
- Assume 1st load takes 8 clocks (L1 cache miss), 2nd load takes 1 clock (hit)
- To be clear, will show clocks for SUBI, BNEZ
  - Reality: integer instructions ahead of Fl. Pt. Instructions
- Show 2 iterations

100

Loop Example

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1			
1	MULTD	F4	F0	F2			
1	SD	F4	0	R1			
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

Iter-ation Count

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Added Store Buffers

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

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

Instruction Loop

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
0	80	Fu									

Value of Register used for address, iteration control

101

Loop Example Cycle 1

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
1	80	Fu	Load1								

102

Loop Example Cycle 2

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
2	80	Fu	Load1		Mult1						

103

Loop Example Cycle 3

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
3	80	Fu	Load1		Mult1						

- **Implicit renaming sets up data flow graph**

104

Loop Example Cycle 4

Instruction status:

ITER	Instruction	j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

					S1	S2	RS
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
4	80	Fu	Load1		Mult1						

- Dispatching SUBI Instruction (not in FP queue)

105

Loop Example Cycle 5

Instruction status:

ITER	Instruction	j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

					S1	S2	RS
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
5	72	Fu	Load1		Mult1						

- And, BNEZ instruction (not in FP queue)

106

Loop Example Cycle 6

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
6	72	Fu	Load2		Mult1						

- Notice that F0 never sees Load from location 80

107

Loop Example Cycle 7

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
7	72	Fu	Load2		Mult2						

- Register file completely detached from computation
- First and Second iteration completely overlapped

108

Loop Example Cycle 8

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
8	72	Fu	Load2		Mult2						

109

Loop Example Cycle 9

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
9	72	Fu	Load2		Mult2						

- Load1 completing: who is waiting?
- Note: Dispatching SUBI

110

Loop Example Cycle 10

Instruction status:

ITER	Instruction	j	k	Issue	Exec Comp	Write Result		Busy	Addr	Fu
1	LD F0	0	R1	1	9	10	Load1	No		
1	MULTD F4	F0	F2	2			Load2	Yes	72	
1	SD F4	0	R1	3			Load3	No		
2	LD F0	0	R1	6	10		Store1	Yes	80	Mult1
2	MULTD F4	F0	F2	7			Store2	Yes	72	Mult2
2	SD F4	0	R1	8			Store3	No		

Reservation Stations:

					S1	S2	RS				
Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	Add1	No						LD	F0	0	R1
	Add2	No						MULTD	F4	F0	F2
	Add3	No						SD	F4	0	R1
4	Mult1	Yes	Multd	M[80]	R(F2)			SUBI	R1	R1	#8
	Mult2	Yes	Multd		R(F2)	Load2		BNEZ	R1	Loop	←

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
10	64	Fu	Load2		Mult2						

- Load2 completing: who is waiting?
- Note: Dispatching BNEZ

111

Loop Example Cycle 11

Instruction status:

ITER	Instruction	j	k	Issue	Exec Comp	Write Result		Busy	Addr	Fu
1	LD F0	0	R1	1	9	10	Load1	No		
1	MULTD F4	F0	F2	2			Load2	No		
1	SD F4	0	R1	3			Load3	Yes	64	
2	LD F0	0	R1	6	10	11	Store1	Yes	80	Mult1
2	MULTD F4	F0	F2	7			Store2	Yes	72	Mult2
2	SD F4	0	R1	8			Store3	No		

Reservation Stations:

					S1	S2	RS				
Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	Add1	No						LD	F0	0	R1 ←
	Add2	No						MULTD	F4	F0	F2
	Add3	No						SD	F4	0	R1
3	Mult1	Yes	Multd	M[80]	R(F2)			SUBI	R1	R1	#8
4	Mult2	Yes	Multd	M[72]	R(F2)			BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
11	64	Fu	Load3		Mult2						

- Next load in sequence

112

Loop Example Cycle 12

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	Multd	M[80]	R(F2)		
3	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
12	64	Fu	Load3		Mult2						

- **Why not issue third multiply?**

113

Loop Example Cycle 13

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	Multd	M[80]	R(F2)		
2	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
13	64	Fu	Load3		Mult2						

- **Why not issue third store?**

114

Loop Example Cycle 14

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	Multd	M[80]	R(F2)		
1	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
14	**64**	*Fu*	Load3		Mult2						

- **Mult1 completing. Who is waiting?**

115

Loop Example Cycle 15

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	**64**	*Fu*	Load3		Mult2						

- **Mult2 completing. Who is waiting?**

116

Loop Example Cycle 16

Instruction status:

ITER	Instruction	j	k	Issue	Exec Comp	Write Result		Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2	14	15	Load2	No		
1	SD	F4	0	R1	3			Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	Yes	80	[80]*R2
2	MULTD	F4	F0	F2	7	15	16	Store2	Yes	72	[72]*R2
2	SD	F4	0	R1	8			Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
16	64	Fu	Load3		Mult1						

Loop Example Cycle 17

Instruction status:

ITER	Instruction	j	k	Issue	Exec Comp	Write Result		Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2	14	15	Load2	No		
1	SD	F4	0	R1	3			Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	Yes	80	[80]*R2
2	MULTD	F4	F0	F2	7	15	16	Store2	Yes	72	[72]*R2
2	SD	F4	0	R1	8			Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
17	64	Fu	Load3		Mult1						

Loop Example Cycle 18

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
18	64	Fu	Load3		Mult1						

119

Loop Example Cycle 19

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	19
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8	19	

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	No		
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
19	56	Fu	Load3		Mult1						

120

Loop Example Cycle 20

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	19
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8	19	20

	Busy	Addr	Fu
Load1	Yes	56	
Load2	No		
Load3	Yes	64	
Store1	No		
Store2	No		
Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	Vj (S1)	Vk (S2)	Qj (RS)	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
20	56	Fu	Load1		Mult1						

- **Once again: In-order issue, out-of-order execution and out-of-order completion.**

121

Why can Tomasulo overlap iterations of loops?

- **Register renaming**
  - Multiple iterations use different physical destinations for registers (dynamic loop unrolling).
- **Reservation stations**
  - Permit instruction issue to advance past integer control flow operations
  - Also buffer old values of registers - totally avoiding the WAR stall that we saw in the scoreboard.
- **Other perspective: Tomasulo building data flow dependency graph on the fly.**

122

Tomasulo's scheme offers 2 major advantages

(1) the distribution of the hazard detection logic

- **distributed reservation stations and the CDB**
- **If multiple instructions waiting on single result, & each instruction has other operand, then instructions can be released simultaneously by broadcast on CDB**
- **If a centralized register file were used, the units would have to read their results from the registers when register buses are available.**

(2) the elimination of stalls for WAW and WAR hazards

123

Dynamic Memory Disambiguation

- **WAR and WAW hazards are eliminated by Tomasulo's algorithm by register renaming**
- **Easy to do since the names are exposed**
- **What about if two instructions share the same memory address ?**

E.g. L.D. F1, 40(R6)
S.D F4, 64(R3)

- **What if 40(R6) = 64(R3) ???**

124

Dynamic Memory Disambiguation

L.D. F1, 40(R6)
S.D F4, 64(R3)

- If the load and store are executed out-of-order… WAR hazard

S.D F4, 64(R3)
L.D. F1, 40(R6)

- RAW hazard

125

Dynamic Memory Disambiguation

- Loads/Stores have to wait until any uncompleted Stores/Loads sharing the same effective address that precede that instruction in program order complete
- To detect these hazards, we need to know the effective address of any earlier memory operation
- Solution: perform the EA calculations in program order

126

Computing EAs in Program Order

- E.g. Consider a load
- When the load completes EA calculation, check the address fields of all the active store buffers
- If the load address matches any active store buffer entry, then do not send the load to the load buffer until the conflicting store completes
- Stores are similar except must check both load and store buffers

127

Review Tomasulo

- Reservations stations: *implicit register renaming* to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards
  - Allows loop unrolling in HW
- Not limited to basic blocks (integer units gets ahead, beyond branches)
- Today, helps cache misses as well
  - Don't stall for L1 Data cache miss (insufficient ILP for L2 miss?)
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- 360/91 descendants are Pentium III, 4; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264

128

Speculation

129

Dynamic Scheduling with Hardware Speculation

- What is speculation? (or speculative execution)
- Let's consider dynamic scheduling (Tomasulo) with hardware branch prediction
- Make a branch prediction and *execute the program as if the guess was correct*
  - The speculatively executed sequence of instructions probably includes other branches (which need to be predicted).
  - This is especially true in multiple-issue processors (possibly one branch per clock cycle)
- Need the ability to undo the effects of an incorrectly speculated sequence

130

Dynamic Scheduling with Hardware Speculation

- Dynamic scheduling without speculation only partially overlaps basic blocks
  - It requires that a branch be resolved before executing instructions in the successor basic block
- Speculation allows us to overcome control dependencies (data flow execution)
- To implement speculation we will modify Tomasulo's algorithm

131

Hardware Speculation

- What is needed to speculatively execute a stream of instructions?
- We must avoid updating the state of the processor until we know for sure that an instruction should have been executed (we then say that it is no longer speculative)
- Registers must not be written until an instruction is no longer speculative
  - Rely on forwarding results among instructions
  - The values forwarded might not be correct.

132

Instruction Commit

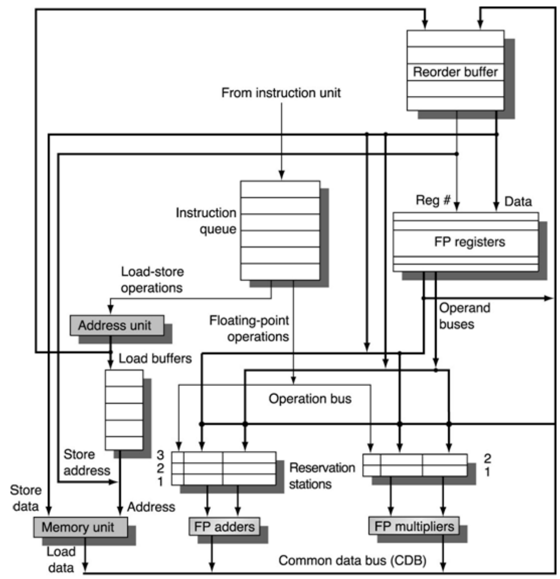
- When we finally know that an instruction is no longer speculative then we allow it to write to the register file or memory
- This extra pipeline stage is called instruction commit
- Key idea: allow instructions to execute *out-of-order* but to commit *in-order*
  - Need to prevent any irrecoverable action (state update, or exception) until the instruction commits
- Instructions may finish execution considerably before they are ready to commit

133

Reorder Buffer

- Need a reorder buffer to hold the results of instructions that have finished execution but have not yet commited
- The ROB also passes results between instructions
  - Register file is updated only when the instruction commits
  - Takes over the role of register renaming from the reservation stations (still need them as buffers between instruction issue and execution)
  - ROB performs the same functionality as the store buffers and it replaces them

134



Four Steps of Speculative Tomasulo Algorithm

1. **Issue—get instruction from Op Queue**
   – If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination
2. **Execution—operate on operands (EX)**
   – When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW
3. **Write result—finish execution (WB)**
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.
4. **Commit—update register with reorder result**
   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer

Reorder Buffer

ROB

Entry	Busy	Instruction	State	Dest	Value
1	no	L.D. F6,34(R2)	commit	F6	Mem[34+Regs[R2]]
2	yes	MUL.D F0,F6,F4	Write result	F0	#1 x Regs[F4]
3	yes	DIV.D F10,F0,F6	Execute	F10	

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Mult1	no	MUL.D	Mem[34+Regs[R2]]	Regs[F4]			#2	
Mult2	yes	DIV.D		Mem[34+Regs[R2]]	#2		#3	

FP Register Status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
Reorder #	2										3
Busy	yes	no	no	no	no	no	no	no	no	no	yes

137

What about Precise Interrupts?

- **Tomasulo had:**

  In-order issue, out-of-order execution, and out-of-order completion

- **Need to "fix" the out-of-order completion aspect so that we can find precise breakpoint in instruction stream.**

138

Relationship between precise interrupts and specultation:

- Speculation is a form of guessing.
- Important for branch prediction:
  - Need to "take our best shot" at predicting branch direction.
- If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly:
  - Need to "fix" the out-of-order completion aspect so that we can find precise breakpoint in instruction stream.
- What about precise exceptions?
  - Need to "fix" the out-of-order completion aspect so that we can find precise breakpoint in instruction stream.

- Technique for both precise interrupts/exceptions and speculation: *in-order completion or commit*

139

Again we take the example of the add-scalar-to-vector loop example. We assume that all integer operations are finished and focus on FP.

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

At this particular time, two complete loops have issued and the first two entries have committed, freeing the ROB entries (fields are left filled for clarity but could be rewritten by other instructions). When the head (currently at entry 3) reaches the branch, if the BNE at entry 5 was mispredicted when issued, then instructions following it are flushed and will never commit.

In essence, the ROB executes **in-order** a simplified version of the original code: it performs just the writes according the actual outcomes of branches. The actual computations are done speculatively according to branch predictions.

140

Multiple-Issue Processors

141

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Basic idea: parallel pipelines.
  - Allow the fetching, issuing, and completion of more than one instruction every clock cycle
- Superscalar: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, Pentium III/4
- Very Long Instruction Words (VLIW): fixed number of instructions (4-16) scheduled by the compiler; put ops into wide templates (TBD)
  - Intel Architecture-64 (IA-64) 64-bit address
    - » Renamed: "Explicitly Parallel Instruction Computer (EPIC)"
  - Will discuss in next chapter

142

Multiple-Issue Processors

Common Name	Issue Structure	Hazard detection	Scheduling	Distinguishing Characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	MIPS, ARM (mainly embedded)
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution (no speculation)	none presently
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Pentium 4, MIPS R12K, IBM Power 5
VLIW	Static	Mostly Software	Static	All hazards determined and indicated by compiler (often implicitly)	C6X
EPIC	Mostly static	Mostly software	Mostly static	Explicit dependences marked by compiler	Itanium

143

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **Superscalar MIPS: 2 instructions, 1 FP & 1 integer**
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

Type	*Pipe Stages*						
Int. instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

- **1 cycle load delay expands to 3 instructions in SS**
  - instruction in right half can't use it, nor instructions in next slot

144

Remember the Unrolled Loop…

```
Loop:L.D    F0,0(R1)
     L.D    F6,-8(R1)
     L.D    F10,-16(R1)
     L.D    F14,-24(R1)
     ADD.D  F4,F0,F2
     ADD.D  F8,F6,F2
     ADD.D  F12,F10,F2
     ADD.D  F16,F14,F2
     S.D    F4,0(R1)
     S.D    F8,-8(R1)
     DADDUI R1,R1,#-32
     S.D    F12,-16(R1)
     BNE    R1,R2,LOOP
     S.D    F16,8(R1) ; 8-32 = -24
```

14 clock cycles, or 3.5 per iteration

145

- **Consider a simple statically scheduled 2-issue MIPS**

	Integer instruction	FP instruction
Loop	L.D F0,0(R1)	
	L.D F6,-8(R1)	
	L.D F10,-16(R1)	ADD.D F4,F0,F2
	L.D F14,-24(R1)	ADD.D F8,F6,F2
	L.D F18,-32(R1)	ADD.D F12,F10,F2
	S.D F4,0(R1)	ADD.D F16,F14,F2
	S.D F8,-8(R1)	ADD.D F20,F18,F2
	S.D F12,-16(R1)	
	DADDUI R1,R1,#-40	
	S.D F16,16(R1)	
	BNE R1,R2,Loop	
	S.D F20,8(R1)	

2.4 cc per iteration

146

Multiple Issue Issues

- issue packet: group of instructions from fetch unit that could potentially issue in 1 clock
  - If instruction causes structural hazard or a data hazard either due to earlier instruction in execution or to earlier instruction in issue packet, then instruction does not issue
  - 0 to N instruction issues per clock cycle, for N-issue
- Performing issue checks in 1 cycle could limit clock cycle time: $O(n^2-n)$ comparisons
  - => issue stage usually split and pipelined
  - 1st stage decides how many instructions from within this packet can issue, 2nd stage examines hazards among selected instructions and those already been issued
  - => higher branch penalties => prediction accuracy important
- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations AND No hazards

147

Static Multiple Issue: VLIW

- Recall superscalar multiple-issue processors:
  - Decide how many instructions to issue on-the-fly
- Statically scheduled superscalar:
  - HW to check for dependencies between instructions in a packet and between instructions in a packet and ones already in the pipeline
- What if we do the dependence checking in the compiler?
  - Format an instruction packet with either no dependencies or at least indicate if they are present
  - Simpler hardware

148

VLIW

- Very long instruction word (VLIW)
- Idea has been around for a long time
- 64 to 128 bit packets
- Drawback: they can be inflexible.
  - Requires recompilation for different versions of the hardware
- Latest versions use software to assist hardware decisions (EPIC → IA-64)

149

The VLIW Idea

- Multiple, independent FUs
- Find independent operations and package them together into a very long instruction word
- Eliminates the expensive hardware that does this in a superscalar
- Superscalar processors are especially expensive for wide issue widths (e.g > 4) so VLIW machines tend to focus on issue widths of > 4

150

VLIW

- E.g. 5-issue VLIW
  - 1 integer (incl. branch)
  - 2 FP
  - 2 memory ref.
- Code must have enough parallelism to fill the operation slots and keep the FUs busy
- Find this parallelism by loop unrolling and scheduling

151

VLIW Example

Mem Ref 1	Mem Ref 2	FP op1	FP op2	Int. op/Branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-24(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-32(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D. F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

- 9 cycles
- 23 operations
- 2.5 operations / cycle
- Efficiency (percent of available slots used) = 60%
- Large number of registers used !

152

VLIW Issues

- **Increased code size**
  - Need to aggressively unroll loops
  - Waste bits whenever instructions are not full
  - Use clever encoding or compression
- **Limitations of lock-step operation**
  - No hazard detection h/w
  - A stall in one FU must stall the whole processor (can't predict cache stalls)
  - Recent processors relax this and use h/w to allow unsynchronized execution
- **Binary code compatibility**
  - Different pipeline organizations require different code (i.e. more FUs)
  - One solution: object code translation (Crusoe: rapidly developing)
  - Another solution: relax this approach (IA-64)

153

Dynamic scheduling approach:

Similarly the Tomasulo algorithm can be applied to the superscalar case. Instead of issuing one instruction per clock cycle, two or more can be issued. Here is an example for the scalar-add-to-vector loop for dual issue. The CC# when events occur are indicated.

Iter.	Instruction	Issue	Exec	Mem	Write CDB	Comment
1	L.D F0,0(R1)	1	2	3	4	
1	ADD.D F4,F0,F2	1	5		8	Wait for L.D
1	S.D F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU R1,R1,-8	2	4		5	Wait for result
1	BNE R1,R2,L	3	6			Wait for DADDIU
2	L.D F0,0(R1)	4	7	8	9	Wait for BNE
2	ADD.D F4,F0,F2	4	10		13	Etc.
2	S.D F4,0(R1)	5	8	14		
2	DADDIU R1,R1,-8	5	9		10	
2	BNE R1,R2,L	6	11			

Quite complex to analyze: many things happen simultaneously.

154

Multiple Issue with Speculation

No speculation:

Iter.	Instruction	Issue	Exec	Mem	Write CDB	Comment
1	LD R2,0(R1)	1	2	3	4	
1	DADDIU R2,R2,#1	1	5		6	Wait for LD
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,-8	2	3		4	Executes directly
1	BNE R1,R2,L	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Etc.
2	SD R2,0(R1)	5	9	13		
2	DADDIU R1,R1,-8	5	8		9	
2	BNE R1,R2,L	6	13			

Speculation:

Iter.	Instruction	Issue	Exec	Read Acces	Write CDB	Commits	Comment
1	LD R2,0(R1)	1	2	3	4	5	
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LD
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,-8	2	3		4	8	Commit in order
1	BNE R1,R2,L	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No delay
2	DADDIU R2,R2,#1	4	8		9	10	Etc.
2	SD R2,0(R1)	5	6			10	
2	DADDIU R1,R1,-8	5	6		7	11	
2	BNE R1,R2,L	6	10			11	

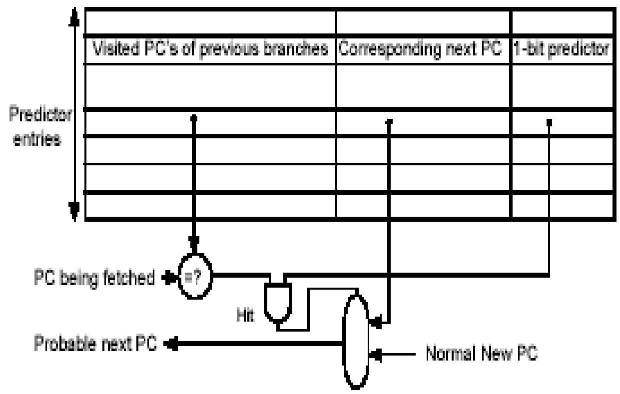
155

High Performance Instruction Delivery

- **Delivering instructions becomes bottleneck, especially in multiple-issue processors**
- **Have to go beyond simple branch prediction**
- **Classic 5-stage pipeline: branch target address and branch direction (outcome) are known early (in ID)**
  - **1 cycle branch delay**
- **Predictors don't give much benefit for this pipeline unless they can give the prediction in the IF stage**
- **Seems impossible: don't even know the instruction yet!**

156

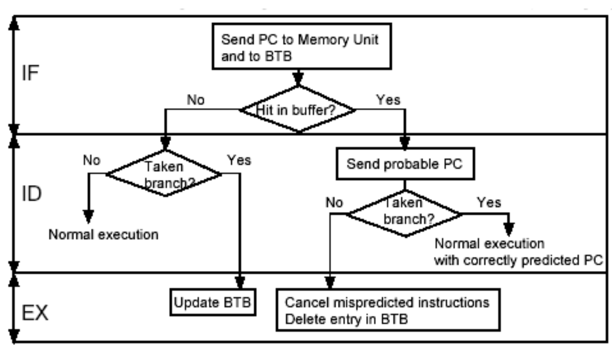
Branch Target Buffer



- Table look up can be done in hardware for small tables
- Usually, only store predicted taken branches in BTB

157

Branch Target Buffer



Example of cost in CC for all cases. This, plus statistics allows to predict speedup.

Hit	Prediction	Outcome	Penalty
yes	taken	taken	0: correct instruction fetched next clock cycle
yes	taken	not taken	2: 1 CC to updtate buffer + 1 CC to restart fetching
no	don't care	taken	2: 1 CC
no	don't care	not taken	0: correct instruction getched next clock cycle

158

Branch Folding

- Next step in this idea is to store the target instruction (instead of just it's address)
- Works perfectly for jumps (unconditional branches) – eliminates them completely (negative penalty!)

159

Integrated Instruction Fetch Units

- Fetching instructions becomes the bottleneck in multiple-issue processors
- Integrated instruction fetch/prediction unit
- Instruction prefetch
  - Fetch ahead several instructions (Chapter 5)

160

Return Address Predictors

- For the procedure call instruction, the return PC is typically stored in a stack in memory
- Instead of loading the return address from memory, some processors provide for a small buffer of the 8-16 most recent return addresses
- Just the knowledge of the PC of a return instruction provides the return address directly without decoding.

161

Advanced Software Approaches

(Appendix G.1,G.2,G.3)

Omit material in G.3 after software pipelining

162

Advanced Compiler Support

- We will study techniques used by modern compilers such as `gcc`
- Dependencies: true and name
- This concept also applies to high-level code
- Compilers can detect parallelism in high-level code that hardware would be blind to

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

163

Loop-Carried Dependencies

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

- If data accesses in an iteration depend on data values produced in earlier iterations we say there is a loop-carried dependence
- This is a parallel loop since there are no loop-carried dependencies.
  - Except for the "induction variable" *i*, but this can be recognized and eliminated (e.g. loop unrolling)

164

Detecting and Exposing Loop-Level Parallelism

- Inspect the code to detect name and data dependencies
- Name dependencies can be eliminated by using more storage ("software renaming")
  - Left with a chain of data dependencies
- If the data dependency chain can be broken, then the loop has some parallelism
- If all data dependencies are within one iteration, the loop is parallel

165

Loop-Carried Dependencies

- Dependencies can exist between statements in a block or across blocks
- Example: recurrences
  - A variable is defined based on the value of that variable in an earlier iteration

e.g.

```
for (i=0;i<=100;++i)
    y[i] = y[i-5] + y[i]
```

Carries a dependency with a dependence distance of 5

166

Finding Dependencies in Loops

- Need to analyze memory references to look for ones that refer to the same addresses
- Difficult in the general case

e.g. `X[Y[i]]`

167

Finding Dependencies in Loops

- Consider finding dependencies in the case when the array indices are "affine"
- An affine index has the form $ai + b$ where i is the loop index and a and b are constants
- To detect a dependence, we need to determine if two affine array indices are equal. i.e

$$ai + b = cj + d$$

168

GCD Test

- **A sufficient test to test for the *absence* of a dependency is the GCD test:**
- **for references `ai + b` and `cj + d`, if a loop dependency exists, then GCD(*c*,*a*) divides (*d*-*b*)**
  - **x *divides* y if y/x is an integer and there is no remainder**
- **Therefore, do the GCD test. If GCD(*c*,*a*) does not divide *d*-*b* then there is no dependency.**
  - **However, the case exists where GCD(*c*,*a*) divides *d*-*b* and there is still no dependency. (because the loop bounds are not considered)**

169

Examples of GCD Test

```
for(i=1;i<=100;++1)
  x[2i+3] = x[2i] + 1.0
```

- **GCD(2,2) does not divide -3**
  - **No dependency is possible**

```
for(i=1;i<=100;++1)
  x[2i+3] = x[2i+1] + 1.0
```

- **2 divides -2**
  - **dependency is possible**
- **In general, deciding if a dependency definitely exists requires an algorithm with an exponential number of steps ("NP-complete") and is not practical**
  - **A few important sub cases are implemented in modern compilers**

170

Classifying Dependencies

- In addition to detecting the presence of dependencies, compilers want to classify the type of dependencies
- E.g. Find the dependencies in:

```
for(i=1;i<=100;i=i+1){
  Y[i] = X[i] / c    /* S1 */
  X[i] = X[i] + c    /* S2 */
  Z[i] = Y[i] + c    /* S3 */
  Y[i] = c - Y[i]    /* S4 */
}
```

True dependence

Antidependence

Output dependence

Example cont'd

```
for (i=1;i<100;i=i+1){
  /* Y renamed to T to remove o.d. */
  T[i] = X[i] / c;
  /* X renamed to U to remove a.d. */
  U[i] = X[i] + c;
  /* Y renamed to T to remove a.d. */
  Z[i] = T[i] + c;
  Y[i] = c - T[i];
}
```

- Second statement is now independent
- Third and fourth only dependent on first

Compiler Loop-Level Transformations

- Transform this loop to make it parallel

```
for (i=1; i < 100; i++) {
 a[i] = b[i] + c[i];     /* S1 */
 b[i] = a[i] + d[i];     /* S2 */
 a[i+1] = a[i] + e[i];  /* S3 */
}
```

173

Dependence Analysis

```
for (i=1; i < 100; i++) {
 a[i] = b[i] + c[i];     /* S1 */
 b[i] = a[i] + d[i];     /* S2 */
 a[i+1] = a[i] + e[i];  /* S3 */
}
```

true data dependency (not loop-carried)

Output dependency (loop-carried)

true data dependency (loop-carried)

Antidependency (not loop-carried)

174

Dependence Analysis

```
a[1] = b[1] + c[1];   /* S1 */
b[1] = a[1] + d[1];   /* S2 */
a[2] = a[1] + e[1];   /* S3 */
a[2] = b[2] + c[2];   /* S1 */
b[2] = a[2] + d[2];   /* S2 */
a[3] = a[2] + e[2];   /* S3 */
a[3] = b[3] + c[3];   /* S1 */
b[3] = a[3] + d[3];   /* S2 */
a[4] = a[3] + e[3];   /* S3 */
…
```

- S3 does no useful work as its result is overwritten by S1 (except on last iteration)

175

Remove S3

```
for (i=1; i < 100; i++) {
  a[i] = b[i] + c[i];    /* S1 */
  U[i] = a[i] + d[i];    /* S2 */
}
a[100] = a[99] + e[99];
```

- Remove antidependence by software renaming
- No loop carried dependencies (parallel loop)

176

Another Example of LLP

```
for (i=1; i < 100; i++) {
 a[i] = a[i] + b[i];    /* S1 */
 b[i+1] = c[i] + d[i];  /* S2 */
}
```

True Dep (loop carried)

- No dependence from S1 to S2
- Can this loop be made parallel?
- No cycles in the dependencies, so yes!

177

Transformed Parallel Loop

```
a[1] = a[1] + b[1]
for (i=1; i <= 99; i++) {
 b[i+1] = c[i] + d[i];
 a[i+1] = a[i+1] + b[i+1];
}
b[101] = c[100] + d[100]
```

178

Algebraic Optimization of Recurrences

- E.g. `sum = sum + x;`

- Unroll a loop with this recurrence 5 times

  `sum = sum + x1 + x2 + x3 + x4 + x5;`
  - 5 dependent operations

- Algebraic optimization

```
sum = ((sum + x1) + (x2 + x3)) + (x4 + x5)
```

  - 3 dependent operations

179

Arithmetic Techniques

- Transformations based on associative and commutative properties of arithmetic
  - not true for limited range and precision, so be careful…
  - Compilers usually will not do these unless explicitly enabled

180

Back Substitution

- E.g. replace

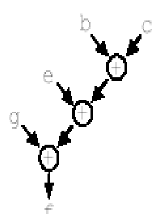
```
DADDUI R1,R2,#4    /* a = b + 4 */
DADDUI R1,R1,#4   /* a = a + 4 */
```

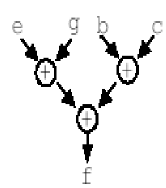
with

```
DADDUI R1,R2,#8    /* a = b + 8 */
```

181

Tree Height Reduction






```
ADD R1,R2,R3   // a=b+c
ADD R4,R1,R6   // d=a+e
ADD R8,R4,R7   // f=d+g
```

Three clock cycles

```
ADD R1,R2,R3   // a=b+c
ADD R4,R6,R7   // d=e+g
ADD R8,R1,R4   // f=a+d
```

Two clock cycles

This applies to cases such as

`sum = sum + x1 + x2 + x4 + x5 = ((sum + x1) + (x2 + x3)) + (x4 + x5),`

etc. The goal of all these transformations is to reduce unnecessary dependencies.

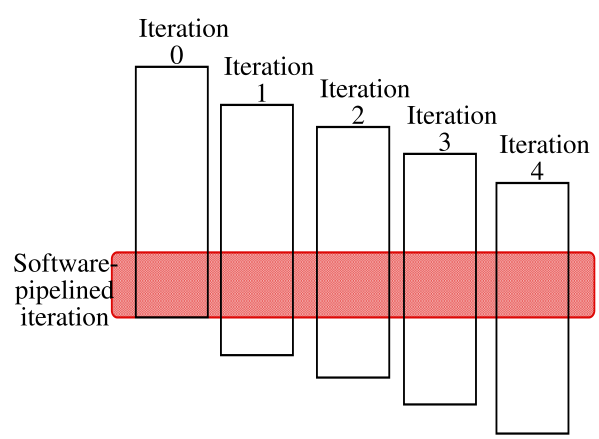
182

Software Pipelining

- The general idea of these optimizations is to uncover long sequences of statements without control statements
- Reorganize loops to interleave instructions from different iterations
  - This is the software counterpart to what Tomasulo's algorithm does in hardware
- Dependent instructions within a single loop iteration are then separated from one another by an entire loop body
  - Increases possibilities of scheduling without stalls

183

Software Pipelining



184

Software Pipelining Example

```
Loop:     L.D.      F0,0(R1)
          ADD.D     F4,F0,F2
          S.D       F4,0(R1)
          DADDUI    R1,R1,#-8
          BNE       R1,R2,LOOP
```

- 10 cycles

Step 1: Symbolic Loop Unrolling

```
ITER i    L.D.      F0,0(R1)
          ADD.D     F4,F0,F2
          S.D       F4,0(R1)
ITER i+1  L.D.      F0,0(R1)
          ADD.D     F4,F0,F2
          S.D       F4,0(R1)
ITER i+2  L.D.      F0,0(R1)
          ADD.D     F4,F0,F2
          S.D       F4,0(R1)
```

Step 2: Select Instructions from Different Iterations

```
ITER i     L.D.      F0,0(R1)
           ADD.D     F4,F0,F2
           S.D       F4,0(R1)
ITER i+1   L.D.      F0,0(R1)
           ADD.D     F4,F0,F2
           S.D       F4,0(R1)
ITER i+2   L.D.      F0,0(R1)
           ADD.D     F4,F0,F2
           S.D       F4,0(R1)
```

187

Step 3. Combine into loop and add init and cleanup code

```
        INIT CODE

Loop:   S.D.       F4,16(R1) ;stores into M[i]
        ADD.D      F4,F0,F2  ;adds to M[i-1]
        L.D        F0,0(R1)  ;loads M[i-2]
        DADDUI     R1,R1,#-8
        BNE        R1,R2,LOOP

        CLEAN UP CODE
```

- 5 clock cycles (assuming DAADUI scheduled before the ADD.D and the L.D is scheduled in the branch delay slot)

188

Software Pipelining

- Advantage: yields shorter code than loop unrolling and uses fewer registers
- Software pipelining is crucial for VLIW processors
  - The above example could be compiled into one instruction
- Often, both software pipelining and loop unrolling are used

189