Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- (Start) Tomasulo Algorithm
- Conclusion

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)

Instruction Level Parallelism

- Instruction-Level Parallelism (ILP): overlap the execution of instructions to improve performance
- Two approaches to exploit ILP:
  1) Rely on hardware to help discover and exploit the parallelism dynamically (e.g., Pentium 4, AMD Opteron, IBM Power), and
  2) Rely on software technology to find parallelism, statically at compile-time (e.g., Itanium 2) – Appendix G
- Next 3 lectures on this topic

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: loop-level parallelism to exploit parallelism among iterations of a loop. E.g., for (i=1; i<=1000; i=i+1) 
  \[ x[i] = x[i] + y[i]; \]

Loop-Level Parallelism

- Exploit loop-level parallelism by “unrolling loop” either by
  1. dynamic via branch prediction or
  2. static via loop unrolling by compiler
  (Another way is vectors, will be covered if time permits)
- Determining instruction dependence is critical to Loop Level Parallelism
- If 2 instructions are parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
- dependent, they are not parallel and must be executed in order, although they may often be partially overlapped
Data Dependence and Hazards

- **InstrJ** is **data dependent** (aka **true dependence**) on **InstrI**:
  1. **InstrJ** tries to read a result produced by **InstrI**
  2. or **InstrJ** is data dependent on **InstrK** which is dependent on **InstrI**
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence \(\Rightarrow\) data dependence in source code \(\Rightarrow\) effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, called a **Read After Write (RAW) hazard**

ILP and Data Dependencies, Hazards

- HW/SW must preserve program order:
  - the order that instructions would execute in if executed sequentially as determined by original source program
  - 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 the order in which results must be calculated
  3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

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
- **InstrJ** writes a register or memory location that **InstrI** reads
  - **I**: \(\text{add } r1, r2, r3\)
  - **J**: \(\text{sub } r4, r1, r3\)
  - **K**: \(\text{mul } r6, r1, r7\)
  - Called an “anti-dependence” by compiler writers. This results from reuse of name “r1”
  - If anti-dependence caused a hazard in the pipeline, called a **Write After Read (WAR) hazard**

Name Dependence #2: Output dependence

- **InstrJ** and **InstrI** write the same register or memory location.
  - **I**: \(\text{sub } r1, r4, r3\)
  - **J**: \(\text{add } r1, r2, r3\)
  - **K**: \(\text{mul } r6, r1, r7\)
  - Called an “output dependence” by compiler writers. This also results from the reuse of name “r1”
  - If output dependence caused a hazard in the pipeline, called a **Write After Write (WAW) hazard**

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 \(p1\).

Control Dependence Could be Ignored

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

• Preserving exception behavior
  \[\Rightarrow\] any changes in instruction execution order must not change how exceptions are raised in program (\(\Rightarrow\) no new exceptions)

• Example:
  
  \[
  \text{DADDU R2, R3, R4} \\
  \text{BEQZ R2, L1} \\
  \text{LW R1, 0(R2)}; \text{no data dependences between BEQZ and LW} \\
  \text{L1:} \\
  \text{– (Assume branches not delayed)} \\
  \text{– Problem with moving LW before BEQZ?} \\
  \text{– A memory protection exception!}
  \]

Preserving Data Flow

• Data flow: actual flow of data values among instructions that produce results and those that consume them
  – branches make data flow dynamic, since the source of data may come from multiple places
  – determine which instruction is supplier of data

• Example:
  
  \[
  \text{DADDU R1, R2, R3} \\
  \text{BEQZ R4, L} \\
  \text{DSUBU R1, R5, R6} \\
  \text{L:} \ldots \\
  \text{OR R7, R1, R8} \\
  \text{– OR depends on DADDU or DSUBU?} \\
  \text{– Must preserve data flow on execution} \\
  \text{– Data dependence alone is not sufficient to preserve correctness}
  \]

Computers in the News

*Again, I'd repeat to you what I've said many times, that if we want the most competitive nation in the world, it will benefit the worker here in America. People have got to understand, when we talk about spending your taxpayers' money on research and development, there is a correlating benefit, particularly to your children. Now, I think a lot of Americans have been made aware of government dollars in coming to them. I don't know if people realize this, but the Internet began as the Defense Department project to improve military communications. In other words, we weren't trying to figure out how to make the Internet faster, but as a result of this sound investment, the Internet came to be.

The Internet has changed us. It's changed the whole world.*

Outline

• ILP
  • Compiler techniques to increase ILP
    – Loop Unrolling
    – Static Branch Prediction
  • Dynamic Branch Prediction
  • Overcoming Data Hazards with Dynamic Scheduling
    – (Start) Tomasulo Algorithm
  • Conclusion

Software Techniques - Example

• This code, add a scalar to a vector:
  
  \[
  \text{for (i=1000; i>0; i=i–1)} \\
  \text{x[i] = x[i] + s;}
  \]

• Assume following latencies for all examples
  – ignore delayed branch in these examples; Branch decision at ID stage

\[
\begin{array}{|c|c|c|}
\hline
\text{Instruction} & \text{Instruction producing result} & \text{Latency}^* \\
\text{Instruction} & \text{using result} & \text{in clock cycles} \\
\hline
\text{FP ALU op} & \text{Another FP ALU op} & 3 \\
\text{FP ALU op} & \text{Store double} & 2 \\
\text{Load double} & \text{FP ALU op} & 1 \\
\text{Load double} & \text{Store double} & 0 \\
\text{Integer op} & \text{Integer op} & 0 \\
\hline
\end{array}
\]

*Stalls in between, it equals the number of stages after EX when the instruction produces the result.

FP Loop: Where are the Hazards?

• First translate into MIPS code:
  – To simplify, assume 8 is lowest address

  \[
  \text{Loop: L.D F0,0(R1); F0=vector element} \\
  \text{ADD.D F4,F0,F2; add scalar from F2} \\
  \text{S.D 0(R1),F4; store result} \\
  \text{DADDUI R1,R1,-8; decrement pointer 8B (DW)} \\
  \text{BNEI R1,Loop ;branch R1!=zero}
  \]

NOW Handout Page 3
FP Loop Showing Stalls

1 Loop: 
1. `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 0(R1)`, F4 :store result
7. stall
8. stall
9. `BNEZ R1`,Loop :branch R1=zero

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction producing result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Load double</td>
<td>1</td>
</tr>
</tbody>
</table>

• 9 clock cycles: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

1 Loop: 
1. `L.D F0,0(R1)`
2. `DADDUI R1,R1,-8`
3. `ADD.D F4,F0,F2` :add scalar in F2
4. stall
5. stall
6. `S.D 8(R1)`, F4 : altered offset when move DSUBUI
7. `BNEZ R1`,Loop

Swap DADDUI and S.D by changing address of S.D

Unroll Loop Four Times (straightforward way)

1 Loop:
1. `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` :add scalar in F2
6. `ADD.D F8,F6,F2` :add scalar in F2
7. `ADD.D F12,F10,F2` :add scalar in F2
8. `ADD.D F16,F14,F2` :add scalar in F2
9. `S.D 0(R1)`, F4 :store result
10. `S.D -8(R1)`, F8 :store result
11. `S.D -16(R1)`, F12 :store result
12. `S.D -24(R1)`, F16 :store result
13. `DADDUI R1,R1,-32` :alter to 4*8
14. `BNEZ R1`,Loop

27 clock cycles, or 6.75 per iteration (Assumes R1 is multiple of 4)

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:
1. `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` :add scalar in F2
6. `ADD.D F8,F6,F2` :add scalar in F2
7. `ADD.D F12,F10,F2` :add scalar in F2
8. `ADD.D F16,F14,F2` :add scalar in F2
9. `DADDUI R1,R1,-8(R1),F4` :alter to 4*8
10. `BNEZ R1`,Loop

14 clock cycles, or 3.5 per iteration

5 Loop Unrolling Decisions

• 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
3 Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
   - Amdahl’s Law
2. Growth in code size
   - For larger loops, concern it increases the instruction cache miss rate
3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
   - If not be possible to allocate all live values to registers, may lose some or all of its advantage
   - Loop unrolling reduces impact of branches on pipeline; another way is branch prediction

Static Branch Prediction

- Lecture 3 showed scheduling code around delayed branch
- To reorder code around branches, need to predict branch statically when compile
- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% SPEC
- Profile-based prediction:
  - The behavior of branches is often bimodally distributed!
  - An individual branch is often highly biased toward taken or not

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
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be
  - There are a small number of important branches in programs which have dynamic behavior

Outline

- ILP
- Compiler techniques to increase ILP
  - Loop Unrolling
  - Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
  - (Start) Tomasulo Algorithm
- Conclusion

Dynamic Branch Prediction

- Performance = f(accuracy, cost of misprediction)
- 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
- Problem: in a loop, 1-bit BHT will cause two mispredictions:
  - End of loop case, when it exits instead of looping as before
  - Beginning of second time loop execution, when the bit signals taken (predict exit instead of looping)

Dynamic Branch Prediction

- Solution: 2-bit scheme where change prediction only if get misprediction twice
- Red: stop, not taken
- Green: go, taken
- Adds hysteresis to decision making process

NOW Handout Page 5
BHT Accuracy
• Mispredict because either:
  – Wrong guess for that branch
  – Got branch history of wrong branch when index the table
• 4096 entry table:
  – A 4K buffer performs similarly to a buffer with infinite entries
  – Increasing the number of bits (e.g. change 2-bit to n-bit) has little impact
  – Change predictor structure!

Correlated Branch Prediction
• 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
  – In general, \((m,n)\) predictor means record last \( m \) branches to select between \( 2^m \) 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 a \( n \)-bit predictor.

Correlating Branches
\((2,2)\) predictor
  – Behavior of recent branches selects between four predictions of next branch, updating just that prediction
  

Accuracy of Different Schemes

Tournament Predictors
• Multilevel branch predictor
• Use \( n \)-bit saturating counter to choose between predictors
• Usual choice between global and local predictors
  • \( n/m \) means:
    \( n \) – predictor 1
    \( m \) – predictor 2
  • 0 – Incorrect; 1 – correct
  • A predictor must be twice incorrect before we switch to the other one

Tournament Predictors (Alpha 21264)
Tournament predictor using, say, 4K 2-bit counters indexed by local branch address. Chooses between:
• Global predictor
  – 4K entries indexed by history of last 12 branches \( (2^{12} \times 4K) \)
  – Each entry is a standard 2-bit predictor
• Local predictor – two levels
  – Local history table: 1024 10-bit entries recording last 10 branches, indexed by branch address
  – The pattern of the last 10 occurrences of that particular branch used to index table of 1K entries with 3-bit saturating counters
Comparing Predictors (Fig. 2.8)

- Advantage of tournament predictor is able to select the right predictor for a particular branch
  - 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.

Pentium 4 Misprediction Rate
(per 1000 instructions, not per branch)

- 46% misprediction rate per branch SPECint (19% of INT instructions are branch)
- 24% misprediction rate per branch SPECfp (5% of FP instructions are branch)

Branch Target Buffers (BTB)

- Branch target calculation is costly and stalls the instruction fetch.
- BTB stores PCs the same way as caches
- The PC of a branch is sent to the BTB
- When a match is found the corresponding Predicted PC is returned
- If the branch was predicted taken, instruction fetch continues at the returned predicted PC

Dynamic Branch Prediction Summary

- Prediction becoming important part of 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 predictors take insight to next level, by using multiple predictors
  - usually 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 Power5 and Pentium 4
- Branch Target Buffer: include branch address & prediction

Outline

- ILP
- Compiler techniques to increase ILP
  - Loop Unrolling
  - Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
  - (Start) Tomasulo Algorithm
- Conclusion
Advantages of Dynamic Scheduling

- Dynamic scheduling - hardware rearranges the instruction execution to reduce stalls while maintaining data flow and exception behavior.
- It handles cases when dependencies are unknown at compile time.
  - It allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the misses to resolve.
- It allows code that compiled for one pipeline to run efficiently on a different pipeline.
- It simplifies the compiler.
- Hardware speculation, a technique with significant performance advantages, builds on dynamic scheduling (next lecture).

HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stalls 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 (e.g., SUBD).
  - In a dynamically scheduled pipeline, all instructions still pass through issue stage in order (in-order issue).
- Will distinguish when an instruction begins execution and when it completes execution; between 2 times, the instruction is in execution.
- Note: Dynamic execution creates WAR and WAW hazards and makes exceptions harder.

Dynamic Scheduling Step 1

- Simple pipeline had 1 stage to check both structural and data hazards: Instruction Decode (ID), also called Instruction Issue.
- To allow out-of-order execution, 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.

A Dynamic Algorithm: Tomasulo’s

- For IBM 360/91 (before caches!)
  - Long memory latency.
- 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, Pentium 4, AMD Opteron, Power 5, ...

Tomasulo Algorithm

- Key Concepts:
  - Tracking dependences to allow instructions execute as soon as operands are available.
  - Register renaming to avoid WAR and WAW hazards.
- Distributed Control & buffers with Function Units (FU)
  - FU buffers called “reservation stations” (RS); have pending operands.
  - RS instead of register file has two properties: allow distributed hazard detection; bypass registers via a common data bus.
- Registers in instructions replaced by values or pointers to reservation stations (RS); called register renaming.
  - 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.
  - Avoids RAW hazards by executing an instruction only when its operands are available.
  - Load and Stores treated as FUs with RSs as well.
  - Integer instructions can go past branches (predict taken), allowing FP ops beyond basic block in FP queue.

Tomasulo Organization

NOW Handout Page 8
Reservation Station Components

Op: Operation to perform in the unit (e.g., + or –)
Vj, Vk: Value of Source operands
- Store buffers have 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 Qi—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   If reservation station free (no structural hazard), control issues inst & sends operands (renames registers), other wise, keep track of the function unit that produces the result.
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)
   - Write if matches expected Functional Unit source address
   - Does the broadcast

• Example speed:
  3 clocks for Fl .pt. +,-; 10 for * ; 40 clks for /

Tomasulo Example Cycle 1

Tomasulo Example Cycle 2

Tomasulo Example Cycle 3

Note: Can have multiple loads outstanding
Tomasulo Example Cycle 4

**Instruction status:** Execute Write

- Instruction
  - Issue: Comp Result
  - Busy Address

**Reservation Stations:**
- S1
- S2
- RS
- RS

**Register result status:**
- Clock
  - F0
  - F2
  - F4
  - F6
  - F8
  - F10

- Load2 completing; what is waiting for Load2?

2/3/2010

Csci 211 – Lecture 4

55

Tomasulo Example Cycle 5

**Instruction status:** Execute Write

- Instruction
  - Issue
  - Comp Result
  - Busy Address

**Reservation Stations:**
- S1
- S2
- RS
- RS

**Register result status:**
- Clock
  - F0
  - F2
  - F4
  - F6
  - F8
  - F10

- Timer starts down for Add1, Mult1

2/3/2010

Csci 211 – Lecture 4

56

Tomasulo Example Cycle 6

**Instruction status:** Execute Write

- Instruction
  - Issue
  - Comp Result
  - Busy Address

**Reservation Stations:**
- S1
- S2
- RS
- RS

**Register result status:**
- Clock
  - F0
  - F2
  - F4
  - F6
  - F8
  - F10

- Issue ADDD here despite name dependency on F6?

2/3/2010

Csci 211 – Lecture 4

57

Tomasulo Example Cycle 7

**Instruction status:** Execute Write

- Instruction
  - Issue
  - Comp Result
  - Busy Address

**Reservation Stations:**
- S1
- S2
- RS
- RS

**Register result status:**
- Clock
  - F0
  - F2
  - F4
  - F6
  - F8
  - F10

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

2/3/2010

Csci 211 – Lecture 4

58

Tomasulo Example Cycle 8

**Instruction status:** Execute Write

- Instruction
  - Issue
  - Comp Result
  - Busy Address

**Reservation Stations:**
- S1
- S2
- RS
- RS

**Register result status:**
- Clock
  - F0
  - F2
  - F4
  - F6
  - F8
  - F10

2/3/2010

Csci 211 – Lecture 4

59

Tomasulo Example Cycle 9

**Instruction status:** Execute Write

- Instruction
  - Issue
  - Comp Result
  - Busy Address

**Reservation Stations:**
- S1
- S2
- RS
- RS

**Register result status:**
- Clock
  - F0
  - F2
  - F4
  - F6
  - F8
  - F10

2/3/2010

Csci 211 – Lecture 4

60
### Tomasulo Example Cycle 10

**Insslustion status:** Issue Write

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+ R2</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>Load1</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>LD F2 45+ R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>MULTD F0 F2 F4</td>
<td>3</td>
<td>1</td>
<td></td>
<td>Load3</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>DIVD F10 F0 F6</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F6 F8 F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reservation Stations:** S1 S2 RS RS

**Time** | **Name** | **Busy** | **Op** | **Vj** | **Vk** | **Qj** | **Qk**
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

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

2/3/2010  Csci 211 – Lecture 4  62

### Tomasulo Example Cycle 11

**Insslustion status:** Issue Write

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+ R3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>Load1</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>LD F2 45+ R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>MULTD F0 F2 F4</td>
<td>3</td>
<td>1</td>
<td></td>
<td>Load3</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SUBD F8 F6 F2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVD F10 F0 F6</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F6 F8 F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reservation Stations:** S1 S2 RS RS

**Time** | **Name** | **Busy** | **Op** | **Vj** | **Vk** | **Qj** | **Qk**
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

- Add2 (ADDD) completing; what is waiting for it?
- Write result of ADDD here?
- All quick instructions complete in this cycle!

2/3/2010  Csci 211 – Lecture 4  65

### Tomasulo Example Cycle 12

**Insslustion status:** Issue Write

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+ R2</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>Load1</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>LD F2 45+ R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>MULTD F0 F2 F4</td>
<td>3</td>
<td>1</td>
<td></td>
<td>Load3</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SUBD F8 F6 F2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F6 F8 F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reservation Stations:** S1 S2 RS RS

**Time** | **Name** | **Busy** | **Op** | **Vj** | **Vk** | **Qj** | **Qk**
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

2/3/2010  Csci 211 – Lecture 4  63

### Tomasulo Example Cycle 13

**Insslustion status:** Issue Write

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+ R3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>Load1</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>LD F2 45+ R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>MULTD F0 F2 F4</td>
<td>3</td>
<td>1</td>
<td></td>
<td>Load3</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SUBD F8 F6 F2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVD F10 F0 F6</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F6 F8 F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reservation Stations:** S1 S2 RS RS

**Time** | **Name** | **Busy** | **Op** | **Vj** | **Vk** | **Qj** | **Qk**
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

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

2/3/2010  Csci 211 – Lecture 4  66

### Tomasulo Example Cycle 14

**Insslustion status:** Issue Write

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+ R2</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>Load1</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>LD F2 45+ R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>MULTD F0 F2 F4</td>
<td>3</td>
<td>1</td>
<td></td>
<td>Load3</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SUBD F8 F6 F2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVD F10 F0 F6</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F6 F8 F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reservation Stations:** S1 S2 RS RS

**Time** | **Name** | **Busy** | **Op** | **Vj** | **Vk** | **Qj** | **Qk**
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

2/3/2010  Csci 211 – Lecture 4  64

### Tomasulo Example Cycle 15

**Insslustion status:** Issue Write

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+ R3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>Load1</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>LD F2 45+ R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>MULTD F0 F2 F4</td>
<td>3</td>
<td>1</td>
<td></td>
<td>Load3</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SUBD F8 F6 F2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVD F10 F0 F6</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F6 F8 F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reservation Stations:** S1 S2 RS RS

**Time** | **Name** | **Busy** | **Op** | **Vj** | **Vk** | **Qj** | **Qk**
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

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

2/3/2010  Csci 211 – Lecture 4  66
Tomasulo Example Cycle 16

Instruction status: Exec Write

<table>
<thead>
<tr>
<th>Instruction</th>
<th>jk</th>
<th>Issue</th>
<th>Comp</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+ R2</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>Load1 No</td>
</tr>
<tr>
<td>LD F2 45+ R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2 No</td>
</tr>
<tr>
<td>MULTD F0 F2 F4</td>
<td>3</td>
<td>15</td>
<td>16</td>
<td>Load3 No</td>
</tr>
<tr>
<td>DIVD F10 F0 F6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F6 F8 F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Reservation Stations: S1 S2 RS RS

Time Name Busy Address
Add No
Add No
40 Mult No DIVD M*F4 M(A1)

Register result status:
Clock 16 F1

• Just waiting for Mult2 (DIVD) to complete

Faster than light computation (skip a couple of cycles)

Tomasulo Example Cycle 55

Instruction status: Exec Write

<table>
<thead>
<tr>
<th>Instruction</th>
<th>jk</th>
<th>Issue</th>
<th>Comp</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+ R2</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>Load1 No</td>
</tr>
<tr>
<td>LD F2 45+ R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2 No</td>
</tr>
<tr>
<td>MULTD F0 F2 F4</td>
<td>3</td>
<td>15</td>
<td>16</td>
<td>Load3 No</td>
</tr>
<tr>
<td>SUBD F6 F8 F2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>ADDD F6 F8 F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Reservation Stations: S1 S2 RS RS

Time Name Busy Address
Add No
Add No
Mult No DIVD M*F4 M(A1)

Register result status:
Clock 86 F1

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

Tomasulo Example Cycle 56

Instruction status: Exec Write

<table>
<thead>
<tr>
<th>Instruction</th>
<th>jk</th>
<th>Issue</th>
<th>Comp</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+ R2</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>Load1 No</td>
</tr>
<tr>
<td>LD F2 45+ R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2 No</td>
</tr>
<tr>
<td>MULTD F0 F2 F4</td>
<td>3</td>
<td>15</td>
<td>16</td>
<td>Load3 No</td>
</tr>
<tr>
<td>SUBD F6 F8 F2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>ADDD F6 F8 F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Reservation Stations: S1 S2 RS RS

Time Name Busy Address
Add No
Add No
40 Mult No DIVD M*F4 M(A1)

Register result status:
Clock 86 F1

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

Tomasulo Example Cycle 57

Instruction status: Exec Write

<table>
<thead>
<tr>
<th>Instruction</th>
<th>jk</th>
<th>Issue</th>
<th>Comp</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+ R2</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>Load1 No</td>
</tr>
<tr>
<td>LD F2 45+ R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2 No</td>
</tr>
<tr>
<td>MULTD F0 F2 F4</td>
<td>3</td>
<td>15</td>
<td>16</td>
<td>Load3 No</td>
</tr>
<tr>
<td>DIVD F10 F0 F6</td>
<td>5</td>
<td>55</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>ADDD F6 F8 F2</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Reservation Stations: S1 S2 RS RS

Time Name Busy Address
Add No
Add No
Mult No DIVD M*F4 M(A1)

Register result status:
Clock 86 F1

• Just waiting for Mult2 (DIVD) to complete

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
• Other perspective: Tomasulo building data flow dependency graph on the fly
Tomasulo’s scheme offers 2 major advantages

1. 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. Elimination of stalls for WAW and WAR hazards

Tomasulo Drawbacks

- Complexity
  - delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620 in CA:QA 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

And In Conclusion … #1

- Leverage Implicit Parallelism for Performance: Instruction Level Parallelism
- Loop unrolling by compiler to increase ILP
- Branch prediction to increase ILP
- Dynamic HW exploiting ILP
  - Works when can't know dependence at compile time
  - Can hide L1 cache misses
  - Code for one machine runs well on another

And In Conclusion … #2

- Reservations stations: 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)

- Helps cache misses as well

- Lasting Contributions
  - Dynamic scheduling
  - Register renaming

- Load/store disambiguation (perform effective address computation in order)

- 360/91 descendants are Intel Pentium 4, IBM Power 5, AMD Athlon/Opteron, …