Instruction–Level Parallelism
COSC 5351 Advanced Computer Architecture
Slides modified from Hennessy CS252 course slides

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
- 2 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)

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% => 3 to 6 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)
  \[ x[i] = x[i] + y[i] \]

When \( i = 2 \) what happens in the loop?
Does \( i = 3 \) interfere with \( i = 4 \)?

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, to be covered later)
- 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

- **Instr**, is data dependent (aka true dependence) on **Instr**.
  1. **Instr** tries to read operand before **Instr** writes it.
  2. **Instr** is data dependent on **Instr**, which is dependent on **Instr**.
- 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**.

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** writes operand before **Instr** reads it.
- **Instr** writes operand before **Instr** writes it.
- 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**.

Name Dependence #2: Output dependence

- **Instr** writes operand before **Instr** writes it.
- **Instr** writes operand before **Instr** writes it.
- 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**.

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 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.

ILP & Data Dependencies, Hazards

- HW/SW must preserve program order:
  - order instructions would execute in if executed sequentially as determined by original source program.
  - Dependencies are a property of programs.
  - Presence of dependence indicates potential for a hazard, but actual hazard and length of any stall is 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.
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program.
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:
    - (Assume branches not delayed)
  - Problem with moving LW before BEQZ?
    - No data dependence but control dependence
    - What if the value in R2 causes a memory access violation?

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 R5,R5,R6
    - L:
      - OR R7,R1,R8

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

Example

- Example:
  - DADDU R1,R2,R3
    - BEQZ R2,skip
    - DSUBU R4,R5,R6
    - DADDU R5,R4,R9
    - skip: OR R7,R1,R8

  - Suppose we knew R4 was not used after (it was dead)
    - Violating the control dependence would not affect the exception behavior or the data flow

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

- Assume following latencies for all examples
  - Ignore delayed branch in these examples

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in cycles</th>
<th>stalls between in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
FP Loop: Where are the Hazards?
• First translate into MIPS code:
  - To simplify, assume 8 is lowest address
    
    for (i=1000; i>0; i=i-1)
    
    w[i] = w[i] + e;
    Put s in F2
    R1 is element with highest address set R1 such that 8(R1) is
    the last element to operate on (notice book uses R1 and R2)

    Loop: L.D F0,0(R1) ; F0=vector element
    ADD.D F4,F0,F2 ; add scalar from F2
    S.D 0(R1),F4 ; store result
    DADDUI R1,R1,-8 ; decrement pointer R1 / (DW)
    BNEZ R1,Loop ; branch R1!=zero

    Revised FP Loop Minimizing Stalls

    Loop: L.D F0,0(R1) ; F0=vector element
    ADD.D F4,F0,F2 ; add scalar from F2
    S.D 0(R1),F4 ; store result
    DADDUI R1,R1,-8 ; decrement pointer R1 / (DW)
    BNEZ R1,Loop ; branch R1!=zero

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

    Instruction   Instruction   Latency in producing result using result
    FP ALU op     FP ALU op     clock cycles
    FP ALU op     Store double   clock cycles
    Load double   FP ALU op     clock cycles

    Revised FP Loop Minimizing Stalls

    Loop: L.D F0,0(R1) ; F0=vector element
    ADD.D F4,F0,F2 ; add scalar from F2
    S.D 0(R1),F4 ; store result
    DADDUI R1,R1,-8 ; decrement pointer R1 / (DW)
    BNEZ R1,Loop ; branch R1!=zero

    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

    Loop: L.D F0,0(R1) ; F0=vector element
    ADD.D F4,F0,F2 ; add scalar from F2
    S.D 0(R1),F4 ; store result
    DADDUI R1,R1,-8 ; decrement pointer R1 / (DW)
    BNEZ R1,Loop ; branch R1!=zero

    14 clock cycles, or 3.5 per iteration

FP Loop Showing Stalls

Unroll Loop Four Times
(straightforward way)

Loop: L.D F0,0(R1) ; F0=vector element
ADD.D F4,F0,F2 ; add scalar from F2
S.D 0(R1),F4 ; store result
DADDUI R1,R1,-8 ; decrement pointer R1 / (DW)
BNEZ R1,Loop ; branch R1!=zero

Rewrite loop to minimize stalls?

Loop: L.D F0,0(R1) ; F0=vector element
ADD.D F4,F0,F2 ; add scalar from F2
S.D 0(R1),F4 ; store result
DADDUI R1,R1,-8 ; decrement pointer R1 / (DW)
BNEZ R1,Loop ; branch R1!=zero

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

Unrolled Loop That Minimizes Stalls

Loop: L.D F0,0(R1) ; F0=vector element
ADD.D F4,F0,F2 ; add scalar from F2
S.D 0(R1),F4 ; store result
DADDUI R1,R1,-8 ; decrement pointer R1 / (DW)
BNEZ R1,Loop ; branch R1!=zero

14 clock cycles, or 3.5 per iteration
Loop Unrolling

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences.
- These 5 decisions and transformations allow us to unroll:
  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
  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 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

- To reorder code around branches, need to predict branch statically when compiled
- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% SPEC92
- More accurate scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run:

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

Dynamic Branch Prediction

- Performance = f(accuracy, cost of misprediction)
- Branch History Table: Lower bits of PC address index a 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 (avg is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop or next time through code, when it predicts exit instead of looping

Dynamic Branch Prediction

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

- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when indexing the table

- 4096 entry table:

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 \( m \) \( n \)-bit predictors.

Correlating Branches

(2,2) predictor

- Behavior of recent branches selects between four predictions of next branch, updating just that prediction

Tournament Predictors

- Multilevel branch predictor
- Use \( n \)-bit saturating counter to choose between predictors
- Usual choice between global and local predictors

Accuracy of Different Schemes

For SPEC89

- 4096 Entries 2-bit BHT
- Unlimited Entries 2-bit BHT
- 1024 Entries (2,2) BHT

Tournament Predictors

Tournament predictor using, say, 4K 2-bit counters indexed by local branch address. Chooses between:

- Global predictor
  - 4K entries index by history of last 12 branches \( (2^{12} = 4K) \)
  - Each entry is a standard 2-bit predictor

- Local predictor
  - Local history table: 1024 10-bit entries recording last 10 branches, index 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 ability 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.

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 one based on global information and one based on local information, and combining them with a selector
  - In 2006, tournament predictors using ≥ 30K bits are in processors like the Power5 and Pentium 4.

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 unknown at compile time
    - It allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss 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

HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stall to proceed
  - `DIVD F0, F2, F4`
  - `ADDQ F10, F7, F9`
  - `SUBQ F12, F8, F14`
- Enables out-of-order execution and allows out-of-order completion (e.g., `SUBQ`)
  - 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
- 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

- 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:
  - 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

Instructions enter Instruction Q and Issued Q FIFO

Reservation stations hold the op and operands + info for hazard detection and resolution

Allow register renaming

Now Handout Page 8
Reserve Station Components

- **Op**: Operation to perform in the unit (e.g., + or -)
- **Vj, Vk**: Value of Source operands
  - Store buffers V field, result to be stored
- **Qj, Qk**: Reservation stations producing source registers (value to be written)
  - Note: Qj,Qk=0 => ready
- **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.

Three Stages of Tomasulo Algorithm

1. **Issue**—Get instruction from FP Op Queue
   - If reservation station free (no structural hazards), 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
  - If write if matches expected Functional Unit (produces result)
  - Does the broadcast
- **Example speed**:
  - 2 clocks for Fl. pt. +, -, 10 for *; 40 clks for /
### Tomasulo Example Cycle 6

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Exec Write</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+</td>
<td>R2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>Load1</td>
</tr>
<tr>
<td>LD F2 45+</td>
<td>R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
</tr>
<tr>
<td>MULT F0 F2 F4</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>Load3</td>
</tr>
<tr>
<td>SUBD F6 F8 F2</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>Load4</td>
</tr>
<tr>
<td>DIVD F0 F6 F0</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>Load5</td>
</tr>
<tr>
<td>ADDD F6 F8 F7</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>Load6</td>
</tr>
</tbody>
</table>

**Reservation Stations:**

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Qj</th>
<th>KQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>S1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

<table>
<thead>
<tr>
<th>Clock</th>
<th>FU</th>
<th>M(A2)</th>
<th>M(A1)</th>
<th>锰1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>F0</td>
<td>Add2</td>
<td>Add1</td>
<td>Add</td>
</tr>
</tbody>
</table>

**Issue ADDD here despite name dependency on F6?**

---

### Tomasulo Example Cycle 7

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Exec Write</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+</td>
<td>R2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>Load1</td>
</tr>
<tr>
<td>LD F2 45+</td>
<td>R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
</tr>
<tr>
<td>MULT F0 F2 F4</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>Load3</td>
</tr>
<tr>
<td>SUBD F6 F8 F2</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>Load4</td>
</tr>
<tr>
<td>DIVD F0 F6 F0</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>Load5</td>
</tr>
<tr>
<td>ADDD F6 F8 F7</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>Load6</td>
</tr>
</tbody>
</table>

**Reservation Stations:**

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Qj</th>
<th>KQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>S1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

<table>
<thead>
<tr>
<th>Clock</th>
<th>FU</th>
<th>M(A2)</th>
<th>M(A1)</th>
<th>锰1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>F0</td>
<td>Add2</td>
<td>Add1</td>
<td>Add</td>
</tr>
</tbody>
</table>

---

### Tomasulo Example Cycle 8

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Exec Write</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+</td>
<td>R2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>Load1</td>
</tr>
<tr>
<td>LD F2 45+</td>
<td>R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
</tr>
<tr>
<td>MULT F0 F2 F4</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>Load3</td>
</tr>
<tr>
<td>SUBD F6 F8 F2</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>Load4</td>
</tr>
<tr>
<td>ADDD F6 F8 F7</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>Load5</td>
</tr>
<tr>
<td>ADD2</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>Load6</td>
</tr>
</tbody>
</table>

**Reservation Stations:**

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Qj</th>
<th>KQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>S1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

<table>
<thead>
<tr>
<th>Clock</th>
<th>FU</th>
<th>M(A2)</th>
<th>M(A1)</th>
<th>锰1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>F0</td>
<td>Add2</td>
<td>Add1</td>
<td>Add</td>
</tr>
</tbody>
</table>

---

### Tomasulo Example Cycle 9

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Exec Write</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+</td>
<td>R2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>Load1</td>
</tr>
<tr>
<td>LD F2 45+</td>
<td>R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
</tr>
<tr>
<td>MULT F0 F2 F4</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>Load3</td>
</tr>
<tr>
<td>SUBD F6 F8 F2</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>Load4</td>
</tr>
<tr>
<td>ADDD F6 F8 F7</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>Load5</td>
</tr>
<tr>
<td>ADD2</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>Load6</td>
</tr>
</tbody>
</table>

**Reservation Stations:**

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Qj</th>
<th>KQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>S1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

<table>
<thead>
<tr>
<th>Clock</th>
<th>FU</th>
<th>M(A2)</th>
<th>M(A1)</th>
<th>锰1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>F0</td>
<td>Add2</td>
<td>Add1</td>
<td>Add</td>
</tr>
</tbody>
</table>

---

### Tomasulo Example Cycle 10

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Exec Write</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+</td>
<td>R2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>Load1</td>
</tr>
<tr>
<td>LD F2 45+</td>
<td>R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
</tr>
<tr>
<td>MULT F0 F2 F4</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>Load3</td>
</tr>
<tr>
<td>SUBD F6 F8 F2</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>Load4</td>
</tr>
<tr>
<td>DIVD F0 F6 F0</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>Load5</td>
</tr>
<tr>
<td>ADDD F6 F8 F7</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>Load6</td>
</tr>
</tbody>
</table>

**Reservation Stations:**

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Qj</th>
<th>KQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>S1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

<table>
<thead>
<tr>
<th>Clock</th>
<th>FU</th>
<th>M(A2)</th>
<th>M(A1)</th>
<th>锰1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>F0</td>
<td>Add2</td>
<td>Add1</td>
<td>Add</td>
</tr>
</tbody>
</table>

---

### Tomasulo Example Cycle 11

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Exec Write</th>
<th>Issue</th>
<th>Comp Result</th>
<th>Busy</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6 34+</td>
<td>R2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>Load1</td>
</tr>
<tr>
<td>LD F2 45+</td>
<td>R3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Load2</td>
</tr>
<tr>
<td>MULT F0 F2 F4</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>Load3</td>
</tr>
<tr>
<td>SUBD F6 F8 F2</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>Load4</td>
</tr>
<tr>
<td>DIVD F0 F6 F0</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>Load5</td>
</tr>
<tr>
<td>ADDD F6 F8 F7</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>Load6</td>
</tr>
</tbody>
</table>

**Reservation Stations:**

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Qj</th>
<th>KQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>S1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register result status:**

<table>
<thead>
<tr>
<th>Clock</th>
<th>FU</th>
<th>M(A2)</th>
<th>M(A1)</th>
<th>锰1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>F0</td>
<td>Add2</td>
<td>Add1</td>
<td>Add</td>
</tr>
</tbody>
</table>

Add3 (SUBD) completing; what is waiting for it?

---

### Tomasulo Example Cycle 12

Write result of ADDD here?

- All quick instructions complete in this cycle!
Tomasulo Example Cycle 12

Instruction status:

<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+ F2 R2</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>5</td>
<td>7</td>
<td>Load3</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SUBD F8 F6 F0</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>Load4</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>DIVD F10 F0 F6</td>
<td>5</td>
<td>10</td>
<td>11</td>
<td>Load5</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Reservation Stations:

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vl</th>
<th>Qj</th>
<th>Qk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S1</td>
<td>Yes</td>
<td>MULTD M(A2)</td>
<td>R(F4)</td>
<td>M(A1)</td>
<td>Mult1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S2</td>
<td>Yes</td>
<td>DIVD</td>
<td>M(A1)</td>
<td>Mult1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register result status:

<table>
<thead>
<tr>
<th>Clock</th>
<th>FU</th>
<th>M(MA)</th>
<th>M(MA+M)</th>
<th>MBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>F0</td>
<td>F2</td>
<td>F4</td>
<td>F6</td>
</tr>
</tbody>
</table>

Tomasulo Example Cycle 13

Instruction status:

<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+ F2 R2</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>5</td>
<td>7</td>
<td>Load3</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SUBD F8 F6 F0</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>Load4</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>DIVD F10 F0 F6</td>
<td>5</td>
<td>10</td>
<td>11</td>
<td>Load5</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Reservation Stations:

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vl</th>
<th>Qj</th>
<th>Qk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S1</td>
<td>Yes</td>
<td>MULTD M(A2)</td>
<td>R(F4)</td>
<td>M(A1)</td>
<td>Mult1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S2</td>
<td>Yes</td>
<td>DIVD</td>
<td>M(A1)</td>
<td>Mult1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register result status:

<table>
<thead>
<tr>
<th>Clock</th>
<th>FU</th>
<th>M(MA)</th>
<th>M(MA+M)</th>
<th>MBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>F0</td>
<td>F2</td>
<td>F4</td>
<td>F6</td>
</tr>
</tbody>
</table>

Tomasulo Example Cycle 14

Instruction status:

<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+ F2 R2</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>5</td>
<td>7</td>
<td>Load3</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SUBD F8 F6 F0</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>Load4</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>DIVD F10 F0 F6</td>
<td>5</td>
<td>10</td>
<td>11</td>
<td>Load5</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Reservation Stations:

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vl</th>
<th>Qj</th>
<th>Qk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S1</td>
<td>Yes</td>
<td>MULTD M(A2)</td>
<td>R(F4)</td>
<td>M(A1)</td>
<td>Mult1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S2</td>
<td>Yes</td>
<td>DIVD</td>
<td>M(A1)</td>
<td>Mult1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register result status:

<table>
<thead>
<tr>
<th>Clock</th>
<th>FU</th>
<th>M(MA)</th>
<th>M(MA+M)</th>
<th>MBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>F0</td>
<td>F2</td>
<td>F4</td>
<td>F6</td>
</tr>
</tbody>
</table>

Tomasulo Example Cycle 15

Instruction status:

<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+ F2 R2</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>5</td>
<td>7</td>
<td>Load3</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SUBD F8 F6 F0</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>Load4</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>DIVD F10 F0 F6</td>
<td>5</td>
<td>10</td>
<td>11</td>
<td>Load5</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Reservation Stations:

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vl</th>
<th>Qj</th>
<th>Qk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S1</td>
<td>Yes</td>
<td>MULTD M(A2)</td>
<td>R(F4)</td>
<td>M(A1)</td>
<td>Mult1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S2</td>
<td>Yes</td>
<td>DIVD</td>
<td>M(A1)</td>
<td>Mult1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register result status:

<table>
<thead>
<tr>
<th>Clock</th>
<th>FU</th>
<th>M(MA)</th>
<th>M(MA+M)</th>
<th>MBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>F0</td>
<td>F2</td>
<td>F4</td>
<td>F6</td>
</tr>
</tbody>
</table>

Faster than light computation (skip a couple of cycles)

Just waiting for Multi2 (DIVD) to complete
Tomasulo Example Cycle 55

Instruction status: Exec Write
Instruction j k Issue Comp Result Busy Address
LD F6 34+ R2 1 3 4 Load1 No
LD F2 45+ R3 2 4 5 Load2 No
MULTI F0 F2 F4 3 15 16 Load3 No
SUBD F8 F6 F5 4 7 8
DIVD F10 F0 F6 6 10 11

Reservation Stations:
Time Name Busy Op Vj Vk Qi Qk
S1 1 S2 2 RS RS

Register result status:
Clock FU M*F4 M(A2) (M-M+M)(M-M)
57

Tomasulo Example Cycle 56

Instruction status: Exec Write
Instruction j k Issue Comp Result Busy Address
LD F6 34+ R2 1 3 4 Load1 No
LD F2 45+ R3 2 4 5 Load2 No
MULTI F0 F2 F4 3 15 16 Load3 No
SUBD F8 F6 F5 4 7 8
DIVD F10 F0 F6 6 10 11

Reservation Stations:
Time Name Busy Op Vj Vk Qi Qk
S1 1 S2 2 RS RS

Register result status:
Clock FU M*F4 M(A2) (M-M+M)(M-M)
55

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
   - If multiple 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: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

NOW Handout Page 13
Speculation to greater ILP

- Greater ILP: Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct
  - *Speculation* ⇒ fetch, issue, and execute instructions as if branch predictions were always correct
  - *Dynamic scheduling* ⇒ only fetches and issues instructions

- Essentially a data flow execution model: Operations execute as soon as their operands are available

Speculation to greater ILP

- 3 components of HW-based speculation:
  1. Dynamic branch prediction to choose which instructions to execute
  2. Speculation to allow execution of instructions before control dependences are resolved
     - + ability to undo effects of incorrectly speculated sequence
  3. Dynamic scheduling to deal with scheduling of different combinations of basic blocks

Adding Speculation to Tomasulo

- Must separate execution from allowing instruction to finish or “commit”
- This additional step called *instruction commit*
- When an instruction is no longer speculative, allow it to update the register file or memory
- Requires additional set of buffers to hold results of instructions that have finished execution but have not committed
- This reorder buffer (ROB) is also used to pass results among instructions that may be speculated

Add Speculation to Tomasulo

- In Tomasulo’s algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits
  - (we know definitively that the instruction should execute)
- Thus, the ROB supplies operands in interval between completion of instruction execution and instruction commit
  - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo’s algorithm
  - ROB extends architected registers like RS

Reorder Buffer (ROB)

- Each entry in the ROB contains four fields:
  1. Instruction type
     - a branch (has no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, which has register destinations)
  2. Destination
     - Register number (for loads and ALU operations) or memory address (for stores) where the instruction result should be written
  3. Value
     - Value of instruction result until the instruction commits
  4. Ready
     - Indicates that instruction has completed execution, and the value is ready

Reorder Buffer operation

- Holds instructions in FIFO order, exactly as issued
- When instructions complete, results placed into ROB
  - Supplies operands to other instruction between execution complete & commit ⇒ more registers like RS
  - Tag results with ROB buffer number instead of reservation station
- Instructions commit ⇒ values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions
Recall: 4 Steps of Speculative Tomasulo Algorithm

1. **Issue**—get instruction from FP Op Queue if reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called "dispatch")
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 (sometimes called "issue")
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 (sometimes called "dispatch")
FP multipliers dynamically maintain the program order for the computation of exceptions. IBM 360/91 invented "imprecise interrupts" not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a destination field that matches the value of the A field of the load, and these restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data.

**Exceptions and Interrupts**
- IBM 360/91 invented "imprecise interrupts".
  - Computer stopped at this PC; its likely close to this address.
  - Not so popular with programmers.
  - Also, what about Virtual Memory? (Not in IBM 360)
- Technique for both precise interrupts/exceptions and speculation: in-order completion and in-order commit.
  - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly.
  - This is exactly same as need to do with precise exceptions.
- Exceptions are handled by not recognizing the exception until instruction that caused it is ready to commit in ROB.
  - If a speculated instruction raises an exception, the exception is recorded in the ROB.
  - This is why reorder buffers in all new processors.

**Getting CPI below 1**
- CPI ≥ 1 if issue only 1 instruction every clock cycle.
- Multiple-issue processors come in 3 flavors:
  1. statically-scheduled superscalar processors,
  2. dynamically-scheduled superscalar processors, and
  3. VLIW (very long instruction word) processors.
- 2 types of superscalar processors issue varying numbers of instructions per clock:
  - use in-order execution if they are statically scheduled, or
  - Out-of-order execution if they are dynamically scheduled.
- VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with parallelism among instructions explicitly indicated by the instruction (Intel/HP Itanium).
VLIW: Very Large Instruction Word

- Each “instruction” has explicit coding for multiple operations
  - In IA–64, grouping called a “packet”
  - In Transmeta, grouping called a “molecule” (with “atoms” as ops)

- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
  - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

Recall: Unrolled Loop that Minimizes Stalls for Scalar

1 Loop:  
1. L.D F0,D(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 F14,F12,F2  
9. S.D 0(R1),F4  
10. S.D -8(R1),F8  
11. S.D -16(R1),F12  
12. DSUBU1 R1,R1,#32  
13. BNEZ R3,LOOP  
14. S.D 0(R1),F16  

14 clock cycles, or 3.5 per iteration

Problems with 1st Generation VLIW

- Increase in code size
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding

- Operated in lock–step; no hazard detection HW
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might prediction function units, but caches hard to predict

- Binary code compatibility
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code

Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,D(R1)</td>
<td>L.D F6,-8(R1)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F18,-32(R1)</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td>L.D F28,-64(R1)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D 0(R1)</td>
<td>S.D -8(R1),F8</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D -16(R1),F12</td>
<td>S.D -24(R1),F16</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D -32(R1),F20</td>
<td>S.D -40(R1),F24</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D -48(R1),F14</td>
<td>S.D -64(R1),F28</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSUBU1 R1,R1,#48</td>
<td>BNEZ R1,LOOP</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6 in SS)

Intel/HP IA–64 “Explicitly Parallel Instruction Computer (EPIC)”

- IA–64: instruction set architecture
  - 128 64-bit integer regs + 128 82-bit floating point regs
  - Not separate register files per functional unit as in old VLIW

- Hardware checks dependencies
  - interlocks => binary compatibility over time
  - Predicated execution (select 1 out of 64 1-bit flags)
  - => 40% fewer mispredictions?

Itanium™ was first implementation (2001)
- Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 µ process

- 8-wide, 6-stage pipeline at 1666Mhz on 0.13 µ process
  - Caches: 32 KB I, 32 KB D, 128 KB L1i, 128 KB L2D, 9216 KB L3

Increasing Instruction Fetch Bandwidth

- Predicts next instruct address, sends it out before decoding instruction
  - PC of branch sent to BTB
  - When match is found, Predicted PC is returned
  - If branch predicted taken, instruction fetch continues at Predicted PC

Branch Target Buffer (BTB)
**More Instruction Fetch Bandwidth**
- Integrated branch prediction: Branch predictor is part of instruction fetch unit and is constantly predicting branches.
- Instruction prefetch: Instruction fetch units prefetch to deliver multiple instructions per clock, integrating it with branch prediction.
- Instruction memory access and buffering: Fetching multiple instructions per cycle.
  - May require accessing multiple cache blocks (prefetch to hide cost of crossing cache blocks).
  - Provides buffering, acting as on-demand unit to provide instructions to issue stage as needed.

**Value Prediction**
- Attempts to predict value produced by instruction.
  - E.g., Loads a value that changes infrequently.
- Value prediction is useful only if it significantly increases ILP.
  - Focus of research has been on loads; so-so results, no processor uses value prediction.
- Related topic is address aliasing prediction.
  - RAW for load and store or WAW for 2 stores.
- Address alias prediction is both more stable and simpler since need not actually predict the address values, only whether such values conflict.
  - Has been used by a few processors.

**Speculation: Register Renaming vs. ROB**
- Alternative to ROB is a larger physical set of registers combined with register renaming.
  - Extended registers replace function of both ROB and reservation stations.
- Instruction issue maps names of architectural registers to physical register numbers in extended register set.
  - On issue, allocates a new unused register for the destination (which avoids WAW and WAR hazards).
  - Speculation recovery easy because a physical register holding an instruction destination does not become the architectural register until the instruction commits.
- Most Out-of-Order processors today use extended registers with renaming.

**Perspective**
- Interest in multiple-issue because wanted to improve performance without affecting uniprocessor programming model.
- Taking advantage of ILP is conceptually simple, but design problems are amazingly complex in practice.
- Conservative in ideas, just faster clock and bigger.
- Recent Processors (Pentium 4, IBM Power 5, AMD Opteron) have the same basic structure and similar sustained issue rates (3 to 4 instructions per clock) as the 1st dynamically scheduled, multiple-issue processors announced in 1995.
  - Clocks 10 to 20X faster, caches 4 to 8X bigger, 2 to 4X as many renaming registers, and 2X as many load-store units ⇒ performance 8 to 16X.
- Peak vs delivered performance gap increasing.

**In Conclusion ...**
- Interrupts and Exceptions either interrupt the current instruction or happen between instructions.
  - Possibly large quantities of state must be saved before interrupting.
- Machines with precise exceptions provide one single point in the program to restart execution.
  - All instructions before that point have completed.
  - No instructions after or including that point have completed.
- Hardware techniques exist for precise exceptions even in the face of out-of-order execution.
  - Important enabling factor for out-of-order execution.