Introduction to Instruction Level Parallelism (ILP)

- What is ILP?
  - Processor and Compiler design techniques that speed up execution by causing individual machine operations to execute in parallel

- ILP is transparent to the user
  - Multiple operations executed in parallel even though the system is handed a single program written with a sequential processor in mind

- Same execution hardware as a normal RISC machine
  - May be more than one of any given type of hardware

Architectures for Instruction-Level Parallelism

Scalar Pipeline (baseline)
- Instruction Parallelism = D
- Operation Latency = 1
- Peak IPC = 1

Superpipelined Machine

Superpipelined Execution
- IP = D x M
- OL = M minor cycles
- Peak IPC = 1 per minor cycle (M per baseline cycle)

SUCCESSIVE INSTRUCTIONS

TIME IN CYCLES (OF BASELINE MACHINE)
Superscalar Machines

Superscalar (Pipelined) Execution
- **IP** = $D \times N$
- **OL** = 1 baseline cycles
- Peak **IPC** = $N$ per baseline cycle

Superscalar and Superpipelined

Superscalar Parallelism
- Operation Latency: 1
- Issuing Rate: $N$
- Superscalar Degree (SSD): $N$
  (Determined by Issue Rate)

Superpipeline Parallelism
- Operation Latency: $M$
- Issuing Rate: 1
- Superpipelined Degree (SPD): $M$
  (Determined by Operation Latency)

Limitations of Inorder Pipelines
- CPI of inorder pipelines degrades very sharply if the machine parallelism is increased beyond a certain point, i.e. when $N \times M$ approaches average distance between dependent instructions
- Forwarding is no longer effective
  \[ \Rightarrow \text{must stall more often} \]
  - Pipeline may never be full due to frequent dependency stalls?

What is parallelism and where

\[
\begin{align*}
x &= a + b; \\
y &= b \times 2; \\
z &= (x-y) \times (x+y)
\end{align*}
\]
What is Parallelism?

- **Work**
  \[ T_1 \] - time to complete a computation on a sequential system

- **Critical Path**
  \[ T_\infty \] - time to complete the same computation on an infinitely-parallel system

- **Average Parallelism**
  \[ P_{avg} = \frac{T_1}{T_\infty} \]

- For a \( p \) wide system
  \[ T_p \geq \max\{ \frac{T_1}{p}, T_\infty \} \]
  \[ P_{avg} \gg p \Rightarrow T_p \approx \frac{T_1}{p} \]

Example Execution

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Operations Performed</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Unit 1</td>
<td>Integer ALU Operations</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Integer Multiplication</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Loads</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Stores</td>
<td>1</td>
</tr>
<tr>
<td>Integer Unit 2</td>
<td>Integer ALU Operations</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Integer Multiplication</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Loads</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Stores</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Test-and-branch</td>
<td>1</td>
</tr>
<tr>
<td>Branch Unit</td>
<td>Integer ALU Operations</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Integer Multiplication</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Loads</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Stores</td>
<td>1</td>
</tr>
<tr>
<td>Floating-point Unit 1</td>
<td>Floating Point Operations</td>
<td>3</td>
</tr>
<tr>
<td>Floating-point Unit 2</td>
<td>Floating Point Operations</td>
<td>3</td>
</tr>
</tbody>
</table>

ILP: Instruction-Level Parallelism

- ILP is a measure of the amount of inter-dependencies between instructions

- Average ILP = \( \) no. instruction / no. cyc required

  - code1: \( ILP = 1 \) (i.e. must execute serially)
  - code2: \( ILP = 3 \) (i.e. can execute at the same time)
Inter-instruction Dependences

- **Data dependence**
  
  \[
  r_3 \leftarrow r_1 \text{ op } r_2 \quad \text{(RAW)}
  \]

  \[
  r_5 \leftarrow r_3 \text{ op } r_4
  \]

- **Anti-dependence**

  \[
  r_3 \leftarrow r_1 \text{ op } r_2 \quad \text{(WAR)}
  \]

  \[
  r_1 \leftarrow r_4 \text{ op } r_5
  \]

- **Output dependence**

  \[
  r_3 \leftarrow r_1 \text{ op } r_2 \quad \text{(WAW)}
  \]

  \[
  r_5 \leftarrow r_3 \text{ op } r_4
  \]

- **Control dependence**

Scope of ILP Analysis

\[
\begin{align*}
ILP=1 & \quad \{ 
  r_1 \leftarrow r_2 + 1 \\
  r_3 \leftarrow r_1 / 17 \\
  r_4 \leftarrow r_0 - r_3 \\
  r_{11} \leftarrow r_{12} + 1 \\
  r_{13} \leftarrow r_{19} / 17 \\
  r_{14} \leftarrow r_0 - r_20
\} \\
ILP=2 & \quad \{ 
\end{align*}
\]

Out-of-order execution permits more ILP to be exploited

Questions Facing ILP System Designers

- What gives rise to instruction-level parallelism in conventional, sequential programs?
- How is the potential parallelism identified and enhanced, and how much is there?
- What must be done in order to exploit the parallelism that has been identified?
- How should the work of identifying, enhancing and exploiting the parallelism be divided between the hardware and software (the compiler)?
- What are the alternatives in selecting the architecture of an ILP processor?
ILP Processors: Superscalar

Sequential Instructions

Superscalar Processor

Instruction scheduling/parallelism extraction done by hardware

ILP Processors: EPIC/VLIW

Serial Program (C code)

Scheduled Instructions

EPIC Processor

Serial Program

compiler

ILP Architectures

- Between the compiler and the run-time hardware, the following functions must be performed
  - Dependencies between operations must be determined
  - Operations that are independent of any operation that has not yet completed must be determined
  - Independent operations must be scheduled to execute at some particular time, on some specific functional unit, and must be assigned a register into which the result may be deposited.

ILP Architecture Classifications

- Sequential Architectures
  - The program is not expected to convey any explicit information regarding parallelism
- Dependence Architectures
  - The program explicitly indicates dependencies between operations
- Independence Architectures
  - The program provides information as to which operations are independent of one another
Sequential Architecture

- Program contains no explicit information regarding dependencies that exist between instructions
- Dependencies between instructions must be determined by the hardware
  - It is only necessary to determine dependencies with sequentially preceding instructions that have been issued but not yet completed
- Compiler may re-order instructions to facilitate the hardware's task of extracting parallelism

Sequential Architecture Example

- Superscalar processor is a representative ILP implementation of a sequential architecture
  - For every instruction issued by a Superscalar processor, the hardware must check whether the operands interfere with the operands of any other instruction that is either
    (1) already in execution, (2) been issued but waiting for completion of interfering instructions that would have been executed earlier in a sequential program, and (3) being issued concurrently but would have been executed earlier in the sequential execution of the program
  - Superscalar proc. issues multiple inst. in cycle

Sequential Architecture Example

- Superscalar processors attempt to issue multiple instructions per cycle
  - However, essential dependencies are specified by sequential ordering so operations must be processed in sequential order
  - This proves to be a performance bottleneck that is very expensive to overcome
- Alternative to multiple instructions per cycle is pipelining and issue instructions faster

Dependence Architecture

- Compiler or programmer communicates to the hardware the dependencies between instructions
  - removes the need to scan the program in sequential order (the bottleneck for superscalar processors)
- Hardware determines at run-time when to schedule the instruction
Dependence Architecture Example

- Dataflow processors are representative of Dependence architectures
  - Execute instruction at earliest possible time subject to availability of input operands and functional units
  - Dependencies communicated by providing with each instruction a list of all successor instructions
  - As soon as all input operands of an instruction are available, the hardware fetches the instruction
  - The instruction is executed as soon as a functional unit is available
- Few Dataflow processors currently exist

Independence Architecture

- By knowing which operations are independent, the hardware needs no further checking to determine which instructions can be issued in the same cycle
  - The set of independent operations is far greater than the set of dependent operations
    - Only a subset of independent operations are specified
- The compiler may additionally specify on which functional unit and in which cycle an operation is executed
  - The hardware needs to make no run-time decisions

Independence Architecture Example

- EPIC/VLIW processors are examples of Independence architectures
  - Specify exactly which functional unit each operation is executed on and when each operation is issued
  - Operations are independent of other operations issued at the same time as well as those that are in execution
  - Compiler emulates at compile time what a dataflow processor does at run-time

Compiler vs. Processor

**VLIW and Superscalar**

- basic structure of VLIW and superscalar consists of a number of EUs, each capable of parallel operation on data fetched from register file
- VLIW and superscalar require highly multiported register files
  - limit on register ports places inherent limitation on maximum number of EUs

**VLIW & Superscalar-Differences**

- presentation of instructions:
  - VLIW receive multi-operation instructions
  - Superscalar accept traditional sequential stream but can issue more than one instruction
- VLIW needs very long instructions in order to specify what each EU should do
- Superscalar receive stream of conventional instructions

**VLIW & Superscalar-Differences**

- Decode and Issue unit in superscalar issues multiple instructions for the EUs
- VLIW expect dependency free code whereas superscalar typically do not expect this. Superscalars cope with dependencies using hardware

**Instruction Scheduling**

- dependencies must be detected and resolved
- *static*: accomplished by compiler which avoids dependencies by rearranging code
- *dynamic*: detection and resolution performed by hardware. Processor typically maintains issue window (prefetched inst) and execution window (being executed). Check for dependencies in issue window.
Instruction Scheduling: The Optimization Goal

- Given a source program $P$, schedule the instructions so as to minimize the overall execution time on the functional units in the target machine.

EPIC/VLIW vs Superscalar: Summary

- In EPIC processors
  - compiler manages hardware resources
  - synergy between compiler and architecture is key
  - some compiler optimizations will be covered in depth

- In Superscalar processors
  - architecture is "self-managed"
  - notably instruction dependence analysis and scheduling done by hardware

Next...

- Basic ILP techniques: dependence analysis, simple code optimization
- Superscalar Processors/ Dynamic ILP
  - Branches
  - scheduling algorithms implemented in hardware
- EPIC Processors
  - Intel IA64 family
  - compiler optimizations needed
- Overview of Compiler Optimization

Superscalar Processors
Superscalar Terminology

• Basic
  Superscalar: Able to issue > 1 instruction / cycle
  Superpipelined: Deep, but not superscalar pipeline.
  E.g., MIPS R5000 has 8 stages
  Branch prediction: Logic to guess whether or not branch will be taken, and possibly branch target

• Advanced
  Out-of-order: Able to issue instructions out of program order
  Speculation: Execute instructions beyond branch points, possibly nullifying later
  Register renaming: Able to dynamically assign physical registers to instructions
  Retire unit: Logic to keep track of instructions as they complete.

Adding Advanced Features

• Out Of Order Issue
  - Can start y as soon as adder available
  - Must hold back z until $f10 not busy & adder available

v: addt $f2, $f4, $f10
w: mult $f10, $f6, $f10
x: addt $f10, $f8, $f12
y: addt $f4, $f6, $f4
z: addt $f4, $f8, $f10

• With Register Renaming

v: addt $f2, $f4, $f10a
w: mult $f10a, $f6, $f10a
x: addt $f10a, $f8, $f12
y: addt $f4, $f6, $f4
z: addt $f4, $f8, $f14

Superscalar Execution Example

Single Order, Data Dependence – In Order

• Assumptions
  - Single FP adder takes 2 cycles
  - Single FP multiplier takes 5 cycles
  - Can issue add & multiply together

(In order) – Must issue in-order

(Single adder, data dependence)

v: addt $f2, $f4, $f10
w: mult $f10, $f6, $f10
x: addt $f10, $f8, $f12
y: addt $f4, $f6, $f4
z: addt $f4, $f8, $f10

Flow Path Model of Superscalars
Superscalar Pipeline Design

Instruction Flow
- Fetch
- Decode
- Dispatch
- Issuing Buffer
- Execute
- Complete
- Store Buffer
- Retire

Data Flow
- Instruction Buffer
- Dispatch Buffer
- Issue Buffer
- Completion Buffer

Inorder Pipelines
- Intel i486
- Intel Pentium

Out-of-order Pipelining 101
- Program Order
- Out-of-order WB

What is the value of F1? WAH!!!
In-order Issue into Diversified Pipelines

- Issue stage needs to check:
  1. Structural Dependence
  2. RAW Hazard
  3. WAW Hazard
  4. WAR Hazard

Superscalar Processors

- Tasks:
  - parallel decoding
  - superscalar instruction issue
  - parallel instruction execution
    - preserving sequential consistency of exception processing
    - preserving sequential consistency of exec.

Parallel Execution

- when instructions executed in parallel they will finish out of program order
  - unequal execution times
- specific means needed to preserve logical consistency
  - preservation of sequential consistency
- exceptions during execution
  - preservation seq. consistency exception proc.
- finishing out of order can be avoided with multiple EU -- how?

More Hardware Features to Support ILP

- Pipelining
  - Advantages
    » Relatively low cost of implementation - requires latches within functional units
    » With pipelining, ILP can be doubled, tripled or more
  - Disadvantages
    » Adds delays to execution of individual operations
    » Increased latency eventually counterbalances increase in ILP
• Additional Functional Units
  – Advantages
    » Does not suffer from increased latency bottleneck
  – Disadvantages
    » Amount of functional unit hardware proportional to degree of parallelism
    » Interconnection network and register file size proportional to square of number of functional units

• Instruction Issue Unit
  – Care must be taken not to issue an instruction if another instruction upon which it is dependent is not complete
  – Requires complex control logic in Superscalar processors
  – Virtually trivial control logic in VLIW processors

• Speculative Execution
  – Little ILP typically found in basic blocks
    » A straight-line sequence of operations with no intervening control flow
  – Multiple basic blocks must be executed in parallel
    » Execution may continue along multiple paths before it is known which path will be executed

• Requirements for Speculative Execution
  – Terminate unnecessary speculative computation once the branch has been resolved
  – Undo the effects of the speculatively executed operations that should not have been executed
  – Ensure that no exceptions are reported until it is known that the excepting operation should have been executed
  – Preserve enough execution state at each speculative branch point to enable execution to resume down the correct path if the speculative execution happened to proceed down the wrong one.
**Hardware Features to Support ILP**

- Speculative Execution
  - Expensive in hardware
  - Alternative is to perform speculative code motion at compile time
    » Move operations from subsequent blocks up past branch operations into proceeding blocks
  - Requires less demanding hardware
    » A mechanism to ensure that exceptions caused by speculatively scheduled operations are reported if and only if flow of control is such that they would have been executed in the non-speculative version of the code
    » Additional registers to hold the speculative execution state

**Introduction to S/W Techniques for ILP**

**Instruction Level Parallelism (ILP)**

- ILP: Overlap execution of unrelated instructions
- How to extract parallelism from program?
  - Beyond single block to get more instruction level parallelism
- Who does instruction scheduling and parallelism extraction?
  - Software or Hardware or mix?
  - Superscalar processors require H/W solutions, but can also use some compiler help
- What new hardware features are required to support more ILP?..?
  - Different requirements for Superscalar and EPIC

**ILP Techniques**

- Key issue to worry about is Hazards
  - Control and data
  - Rising out of dependencies
- How to increase ILP
  - Reduce data hazards: RAW, WAR, WAW
  - Handle control hazards better
  - Increase ideal IPC (instructions per cycle)
- Note: Bottom line is how to better schedule instructions
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)

Ideas to Reduce Stalls

<table>
<thead>
<tr>
<th>Technique</th>
<th>Reduces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic scheduling</td>
<td>Data hazard stalls</td>
</tr>
<tr>
<td>Dynamic branch prediction</td>
<td>Control stalls</td>
</tr>
<tr>
<td>Issuing multiple instructions per cycle</td>
<td>Ideal CPI</td>
</tr>
<tr>
<td>Speculation</td>
<td>Data and control stalls</td>
</tr>
<tr>
<td>Dynamic memory disambiguation</td>
<td>Data hazard stalls involving memory</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>Control hazard stalls</td>
</tr>
<tr>
<td>Basic compiler pipeline scheduling</td>
<td>Data hazard stalls</td>
</tr>
<tr>
<td>Compiler dependence analysis</td>
<td>Ideal CPI and data hazard stalls analysis</td>
</tr>
<tr>
<td>Software pipelining and trace scheduling</td>
<td>Ideal CPI and data hazard stalls</td>
</tr>
<tr>
<td>Compiler speculation</td>
<td>Ideal CPI, data and control stalls</td>
</tr>
</tbody>
</table>

Chapter 3

Chapter 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: loop-level parallelism 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

Data Dependence and Hazards

- $\text{Instr}_j$ is data dependent on $\text{Instr}_i$
  - $\text{Instr}_j$ tries to read operand before $\text{Instr}_i$ writes it

\[
\text{I: } \text{add } r1, r2, r3 \\
\text{J: } \text{sub } r4, r1, r3
\]

- or $\text{Instr}_j$ is data dependent on $\text{Instr}_k$ which is dependent on $\text{Instr}_i$
- Caused by a “True Dependence” (compiler term)
- If true dependence caused a hazard in the pipeline, called a **Read After Write (RAW) hazard**
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.

Name Dependence #1: Anti-dependence

- Name dependence: when 2 instructions use the 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.

\[
\begin{align*}
I & : \text{sub } r4, r1, r3 \\
J & : \text{add } r1, r2, r3 \\
K & : \text{mul } r6, r1, r7
\end{align*}
\]

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\(_J\) writes operand before Instr\(_I\) writes it.

\[
\begin{align*}
I & : \text{sub } r1, r4, r3 \\
J & : \text{add } r1, r2, r3 \\
K & : \text{mul } r6, r1, r7
\end{align*}
\]

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

```java
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 exception behavior and data flow

Exception Behavior

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

• Example:

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

• Problem with moving LW before BEQZ?

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:

```java
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
Loop Unrolling: A Simple S/W Technique

- Parallelism within one “basic block” is minimal
  - Need to look at larger regions of code to schedule
- Loops are very common
  - Number of iterations, same tasks in each iteration
- Simple Observation: If iterations are independent, then multiple iterations can be executed in parallel
- Loop Unrolling: Unrolling multiple iterations of a loop to create more instructions to schedule

Example FP Loop: Where are the Hazards?

Loop:  
LD F0,0(R1); F0=vector element  
ADDD F4,F0,F2; add scalar from F2  
SD 0(R1),F4; store result  
SUBI R1,R1,8; decrement pointer 8B (DW)  
BNEZ R1,Loop; branch R1!=zero  
NOP; delayed branch slot

Example FP Loop Showing Stalls

1 Loop:  
LD F0,0(R1); F0=vector element
2 stall
3 ADDD F4,F0,F2; add scalar from F2
4 stall
5 stall
6 SD 0(R1),F4; store result
7 SUBI R1,R1,8; decrement pointer 8B (DW)
8 BNEZ R1,Loop; branch R1!=zero
9 stall; delayed branch slot

 FP Loop Hazards

Loop:  
LD F0,0(R1); F0=vector element  
ADDD F4,F0,F2; add scalar from F2  
SD 0(R1),F4; store result  
SUBI R1,R1,8; decrement pointer 8B (DW)  
BNEZ R1,Loop; branch R1!=zero  
NOP; delayed branch slot

Instruction producing result | Instruction using result | Latency in clock cycles
---|---|---
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
Integer op | Integer op | 0
Revised FP Loop Minimizing Stalls

1. Loop: LD F0,0(R1)
2. stall
3. ADDD F4,F0,F2
4. SUBI R1,R1,8
5. BNEZ R1,Loop ;delayed branch
6. SD #(R1),F4 ;altered when move past SUBI

Swap BNEZ and SD by changing address of SD

Unrolled Loop That Minimizes Stalls

1. Loop: LD F0,0(R1)
2. ADDD F4,F0,F2
3. SD 0(R1),F4 ;drop SUBI & BNEZ
4. LD F6,-8(R1)
5. ADDD F8,F6,F2
6. SD -8(R1),F8 ;drop SUBI & BNEZ
7. LD F10,-16(R1)
8. ADDD F12,F10,F2
9. SD -16(R1),F12 ;drop SUBI & BNEZ
10. LD F14,-24(R1)
11. ADDD F16,F14,F2
12. SD -24(R1),F16
13. SUBI R1,R1,#32 ;alter to 4*8
14. BNEZ R1,LOOP
15. NOP

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration
Assumes R1 is multiple of 4

Compiler Perspectives on Code Movement

- Definitions: compiler concerned about dependencies in program, whether or not a HW hazard depends on a given pipeline
- Try to schedule to avoid hazards
- (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:
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
Where are the data dependencies?

1 Loop: LD F0, 0(R1)
2 ADDD F4, F0, F2
3 SUBI R1, R1, 8
4 BNEZ R1, Loop ; delayed branch
5 SD 8(R1), F4 ; altered when move past SUBI

Compiler Perspectives on Code Movement

• Another kind of dependence called name dependence: two instructions use same name (register or memory location) but don’t exchange data

• Antidependence (WAR if a hazard for HW)
  – Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first

• Output dependence (WAW if a hazard for HW)
  – Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.

Where are the name dependencies?

1 Loop: LD F0, 0(R1)
2 ADDD F4, F0, F2
3 SD 0(R1), F4 ; drop SUBI & BNEZ
4 LD F0, -8(R1)
5 ADDD F4, F0, F2
6 SD -8(R1), F4 ; drop SUBI & BNEZ
7 LD F0, -16(R1)
8 ADDD F4, F0, F2
9 SD -16(R1), F4 ; drop SUBI & BNEZ
10 LD F0, -24(R1)
11 ADDD F4, F0, F2
12 SD -24(R1), F4
13 SUBI R1, R1, #32 ; alter to 4*8
14 BNEZ R1, LOOP
15 NOP

How can remove them?

Called “register renaming”
Compiler Perspectives on Code Movement

• Again Name Dependences are Hard 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

Compiler Perspectives on Code Movement

• Final kind of dependence called control dependence

• Example
  if p1 {S1;};
  if p2 {S2;};
S1 is control dependent on p1 and S2 is control dependent on p2 but not on p1.

Compiler Perspectives on Code Movement

• Two (obvious) constraints on control dependences:
  – An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
  – An instruction that is not control dependent on a branch cannot be moved to after the branch so that its execution is controlled by the branch.

• Control dependencies relaxed to get parallelism; get same effect if preserve order of exceptions (address in register checked by branch before use) and data flow (value in register depends on branch)

Where are the control dependencies?

1 Loop: LD F0,0(R1)
2 ADDD F4,F0,F2
3 SD 0(R1),F4
4 SUBI R1,R1,8
5 BEQZ R1,exit
6 LD F0,0(R1)
7 ADDD F4,F0,F2
8 SD 0(R1),F4
9 SUBI R1,R1,8
10 BEQZ R1,exit
11 LD F0,0(R1)
12 ADDD F4,F0,F2
13 SD 0(R1),F4
14 SUBI R1,R1,8
15 BEQZ R1,exit
....
When Safe to Unroll Loop?

- Example: Where are data dependencies?
  \((A, B, C)\) distinct & nonoverlapping
  ```
  for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i];    /* S1 */
    B[i+1] = B[i] + A[i+1];  /* S2 */
  }
  ```
  1. S2 uses the value, \(A[i+1]\), computed by S1 in the same iteration.
  2. S1 uses a value computed by S1 in an earlier iteration, since
     iteration \(i\) computes \(A[i+1]\) which is read in iteration \(i+1\). The same
     is true of S2 for \(B[i]\) and \(B[i+1]\). This is a "loop-carried dependence": between iterations
- Implies that iterations are dependent, and can't be
  executed in parallel
- Not the case for our prior example; each iteration was
  distinct

HW Schemes: Instruction Parallelism

- Why in HW at run time?
  - Works when can't know real dependence at compile time
  - Compiler simpler
  - Code for one machine runs well on another
- 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 => out-of-order completion
- ID stage checked both for structural
  scoreboard dates to CDC 6600 in 1963

Next...

- How to deal with instruction flow
  - Dynamic Branch prediction
- How to deal with register/data flow
  - Register renaming
- Dynamic branch prediction
- Dynamic scheduling using Tomasulo method