Several Common Compiler Strategies

- Instruction scheduling
- Loop unrolling
- Static Branch Prediction
- Software Pipelining
Basic Instruction Scheduling

- Reschedule the order of the instructions to reduce the hazards

Ex:

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

Loop:
- **L.D** F0, 0(R1)
- **Add.D** F4, F0, F2
- **S.D** F4, 0(R1)
- **DADDUI** R1, R1, #-8
- **BNE** R1, R2, Loop
Example

- Non-scheduled code and execution

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

stall
stall
stall
stall

BNE   R1, R2, Loop

stall

Loop: L.D   F0, 0(R1)
s stall
Add.D   F4, F0, F2
stall
DADDI   R1, R1, #-8
stall
BNE   R1, R2, Loop
stall
Example (cont’d)

- Scheduled code and execution

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

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

- Unrolling the loop bodies to reduce the loop overhead and increase the ILP
- Example:
  - Still use the previous example
Example

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

Loop: L.D F0, 0(R1)
Add.D F4, F0, F2
S.D F4, 0(R1)
L.D F6, -8(R1)
Add.D F8, F6, F2
S.D F8, -8(R1)
L.D F10, -16(R1)
Add.D F12, F10, F2
S.D F12, -16(R1)
L.D F14, -24(R1)
Add.D F16, F14, F2
S.D F16, -24(R1)
DADDUI R1, R1, # -32
BNE R1, F2, loop
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Loop: L.D</th>
<th></th>
<th>Loop: L.D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>F0, 0(R1)</td>
<td></td>
<td>F0, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>Add.D</td>
<td>F4, F0, F2</td>
<td>L.D</td>
<td>F6, -8(R1)</td>
<td>L.D</td>
<td>F10, -16(R1)</td>
</tr>
<tr>
<td>S.D</td>
<td>F4, 0(R1)</td>
<td>L.D</td>
<td>F10, -16(R1)</td>
<td>L.D</td>
<td>F14, -24(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F6, -8(R1)</td>
<td>L.D</td>
<td>F14, -24(R1)</td>
<td>L.D</td>
<td>F16, 8(R1)</td>
</tr>
<tr>
<td>Add.D</td>
<td>F8, F6, F2</td>
<td>Add.D</td>
<td>F4, F0, F2</td>
<td>Add.D</td>
<td>F8, F6, F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F8, -8(R1)</td>
<td>Add.D</td>
<td>F8, F6, F2</td>
<td>Add.D</td>
<td>F12, F10, F2</td>
</tr>
<tr>
<td>L.D</td>
<td>F10, -16(R1)</td>
<td>Add.D</td>
<td>F12, F10, F2</td>
<td>Add.D</td>
<td>F16, F14, F2</td>
</tr>
<tr>
<td>Add.D</td>
<td>F12, F10, F2</td>
<td>Add.D</td>
<td>F16, F14, F2</td>
<td>S.D</td>
<td>F4, 0(R1)</td>
</tr>
<tr>
<td>S.D</td>
<td>F12, -16(R1)</td>
<td>S.D</td>
<td>F8, -8(R1)</td>
<td>S.D</td>
<td>F8, -8(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F14, -24(R1)</td>
<td>S.D</td>
<td>F16, -24(R1)</td>
<td>S.D</td>
<td>F12,16(R1)</td>
</tr>
<tr>
<td>Add.D</td>
<td>F16, F14, F2</td>
<td>DADDUI</td>
<td>R1, R1, #32</td>
<td>BNE</td>
<td>R1, F2, loop</td>
</tr>
<tr>
<td>S.D</td>
<td>F16, -24(R1)</td>
<td>S.D</td>
<td>F12,16(R1)</td>
<td>S.D</td>
<td>F16, 8(R1)</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1, #32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>R1, F2, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Static Branch Prediction

• Predicting the branch taken/untaken
  – Always taken
  – On the basis of branch direction
  – On the basis of profile information
Software Pipelining

- Choosing instructions from different loop iterations
- Example
Loop: L.D         F0, 0(R1)
Add.D     F4, F0, F2
S. D        F4, 0(R1)
DADDUI  R1, R1, #-8
BNE      R1, R2, Loop

Iter1: L.D       F0, 0(R1)
Add.D     F4, F0, F2
S. D        F4, 0(R1)

Iter2: L.D       F0, 0(R1)
Add.D     F4, F0, F2
S. D        F4, 0(R1)

Iter3: L.D       F0, 0(R1)
Add.D     F4, F0, F2
S. D        F4, 0(R1)

Loop: S.D         F4, 16(R1)
Add.D     F4, F0, F2
L. D      F0, 0(R1)
DADDUI  R1, R1, #-8
BNE      R1, R2, Loop
Example (cont’d)

Loop: S.D   F4, 16(R1)
Add.D    F4, F0, F2
L. D     F0, 0(R1)
DADDUI   R1, R1, #8
stall
BNE      R1, R2, Loop
stall

Loop: S.D   F4, 16(R1)
DADDUI   R1, R1, #8
Add.D    F4, F0, F2
BNE      R1, R2, Loop
L. D     F0, 8(R1)
Software Pipelining

• Use loop body to separate the instructions with data dependency
  – Combined loop unrolling and software pipelining
• Less code space
• Can be quite difficult
  – May require significant transformation
  – Trade-offs are complex
  – Register management can be difficult
To Further Exploit the ILP: The VLIW Approach

- **VLIW**: *Very long width instructions*
  - Multiple, independent functional units
  - Multiple operations per instruction
- **Compiler helps to organize the instructions to avoid the hazards**
Example

- Two memory access reference
- Two FP operations
- One integer operations or branch
- Unrolling the loop as necessary
- \( x[i] = x[i] + s \)
<table>
<thead>
<tr>
<th>Mem Ref1</th>
<th>Mem Ref2</th>
<th>FP Op1</th>
<th>FP Op2</th>
<th>Int Op/Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0, 0(R1)</td>
<td>L.D F6, -8(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F10, -16(R1)</td>
<td>L.D F14, -24(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F18, -32(R1)</td>
<td>L.D F22, -40(R1)</td>
<td>Add.D F4, F0, F2</td>
<td>Add. D F8, F6, F2</td>
<td></td>
</tr>
<tr>
<td>L.D F26, -48(R1)</td>
<td></td>
<td>Add.D F12, F10, F2</td>
<td>Add.D F16, F14, F2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Add.D F20, F18, F2</td>
<td>Add.D F24, F22, F2</td>
<td></td>
</tr>
<tr>
<td>S.D F4, 0(R1)</td>
<td>S.D F8, -8(R1)</td>
<td>Add.D F28, F26, F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F12, -16(R1)</td>
<td>S.D F16, -24(R1)</td>
<td></td>
<td></td>
<td>DADDUI, R1, F1, #56</td>
</tr>
<tr>
<td>S.D F20, 24(R1)</td>
<td>S.D F24, 16(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F28, 8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>BNE R1, R2, Loop</td>
</tr>
</tbody>
</table>
Problems for VLIW Approach

• **Technical problems**
  – Increased code size
    • Unrolling loops increases code size
    • “Wasted bits” in the instructions when instructions are not full
    – A stall in a functional unit causes other units to stall

• **Logistical problem**
  – Binary code compatibility
VLIW or EPIC

- **VLIW (very long instruction word):**
  
  Compiler packs a fixed number of instructions into a single VLIW instruction. The instructions within a VLIW instruction are issued and executed in parallel.  
  **Example:** High-end signal processors (TMS320C6201)

- **EPIC (explicit parallel instruction computing):**
  
  Evolution of VLIW  
  **Example:** Intel’s IA-64, exemplified by the Itanium processor
EPIC Processors, Intel's IA-64 ISA and Itanium

- Joint R&D project by Hewlett-Packard and Intel (announced in June 1994)
- This resulted in explicitly parallel instruction computing (EPIC) design style:
  - specifying ILP explicit in the machine code, that is, the parallelism is encoded directly into the instructions similarly to VLIW;
  - a fully predicated instruction set;
  - an inherently scalable instruction set (i.e., the ability to scale to a lot of FUs);
  - many registers;
  - speculative execution of load instructions
EPIC design challenges

- Develop architectures applicable to general-purpose computing
  - Find substantial parallelism in “difficult to parallelize” scalar programs
  - Provide compatibility across hardware generations
  - Support emerging applications (e.g. multimedia)

- Compiler must find or create sufficient ILP

- Combine the best attributes of VLIW & superscalar RISC
  (incorporated best concepts from all available sources)

- Scale architectures for modern single-chip implementation
IA-64 Architecture

- Unique architecture features & enhancements
  - Explicit parallelism and templates
  - Predication, speculation, memory support, and others
  - Floating-point and multimedia architecture
- IA-64 resources available to applications
  - Large, application visible register set
  - Rotating registers, register stack, register stack engine
- IA-32 & PA-RISC compatibility models
Today's Architecture Challenges

• Performance barriers:
  - Memory latency
  - Branches
  - Loop pipelining and call / return overhead

• Headroom constraints:
  - Hardware-based instruction scheduling
    • Unable to efficiently schedule parallel execution
  - Resource constrained
    • Too few registers
    • Unable to fully utilize multiple execution units

• Scalability limitations:
  - Memory addressing efficiency
Intel's IA-64 ISA

- Intel 64-bit Architecture (IA-64) register model:
  - 128 64-bit general purpose registers GR0-GR127 to hold values for integer and multimedia computations
    - each register has one additional NaT (Not a Thing) bit to indicate whether the value stored is valid,
  - 128 82-bit floating-point registers FR0-FR127
    - registers f0 and f1 are read-only with values +0.0 and +1.0,
  - 64 1-bit predicate registers P0-PR63
    - the first register p0 is read-only and always reads 1 (true)
  - 8 64-bit branch registers BR0-BR7 to specify the target addresses of indirect branches
IA-64's Large Register File

**Integer Registers**
- GR0
- GR1
- GR31
- GR32
- GR127
  - NaT
  - 32 Static
  - 96 Stacked, Rotating

**Floating-Point Registers**
- GR0
- GR1
- GR31
- GR32
- GR127
  - 32 Static

**Branch Registers**
- BR0
- BR1
- BR7
  - 63
  - 0

**Predicate Registers**
- PR0
- PR1
- PR15
- PR16
- PR63
  - 16 Static
  - 48 Rotating
Intel's IA-64 ISA

- IA-64 instructions are 41-bit (previously stated 40 bit) long and consist of
  - op-code,
  - predicate field (6 bits),
  - two source register addresses (7 bits each),
  - destination register address (7 bits), and
  - special fields (includes integer and floating-point arithmetic).
- The 6-bit predicate field in each IA-64 instruction refers to a set of 64 predicate registers.
- 6 types of instructions
  - A: Integer ALU ==> I-unit or M-unit
  - I: Non-ALU integer ==> I-unit
  - M: Memory ==> M-unit
  - B: Branch ==> B-unit
  - F: Floating-point ==> F-unit
  - L: Long Immediate ==> I-unit
- IA-64 instructions are packed by compiler into bundles.
Memory Support for High Performance Technical Computing

- Scientific analysis, 3D graphics and other technical workloads tend to be predictable & memory bound
- IA-64 data pre-fetching of operations allows for fast access of critical information
  - Reduces memory latency impact
- IA-64 able to specify cache allocation
  - Cache hints from load / store operations allow data to be placed at specific cache level
  - Efficient use of caches, efficient use of bandwidth
IA Server/Workstation Roadmap

IA-64 starts with Merced processor

All dates specified are target dates provided for planning purposes only and are subject to change.
Itanium

- 64-bit processor ==> not in the Pentium, PentiumPro, Pentium II/III-line
- Targeted at servers with moderate to large numbers of processors
- full compatibility with Intel’s IA-32 ISA
- EPIC (explicitly parallel instruction computing) is applied.
- 6-wide (3 EPIC instructions) pipeline
- 10 stage pipeline
- 4 int, 4 multimedia, 2 load/store, 3 branch, 2 extended floating-point, 2 single-prec. Floating-point units
- Multi-level branch prediction besides predication
- 16 KB 4-way set-associative d- and I-caches
- 96 KB 6-way set-associative L2 cache
- 4 MB L3 cache (on package)
- 800 MHz, 0.18 micro process (at beginning of 2001)
- shipments end of 1999 or mid-2000 or ??
Conceptual View of Itanium
Itanium Processor Core Pipeline

ROT: instruction rotation
pipelined access of the large register file:
    WDL: word line decode:
    REG: register read
DET: exception detection (~retire stage)
Itanium Die Plot
Itanium vs. Willamette (P4)

- Itanium announced with 800 MHz
- P4 announced with 1.2 GHz
- P4 may be faster in running IA-32 code than Itanium running IA-64 code
- Itanium probably won't compete with contemporary IA-32 processors
- but Intel will complete the Itanium design anyway
- Intel hopes for the Itanium successor McKinley which will be out only one year later