VU2 185.324

Compilation Techniques for VLIW Architectures

Dietmar Ebner ebner@complang.tuwien.ac.at
Florian Brandner brandner@complang.tuwien.ac.at

http://complang.tuwien.ac.at/cd/vliw

Last Lecture

- Product life cycle
  - Product volume ("long tail" vs. "head")
- Constraints
  - Performance/power/size
  - Production costs/Development costs
  - Market/Time to market

In Today's Lecture

- VLIW principles
  - ILP
  - VLIW vs. Superscalar
- Instruction Set
  - Execution model
  - Extensions
- Instruction Encoding

Last Lecture (2)

- Characterization
  - Main purpose is not computing
  - Applications (consumer, comm., automotive)
  - Processor architecture
  - Work load
- Compatibility
- Custom solutions (ASIP, DSP, SoC)
VLIW Principles

Expose ILP in the architecture design.

If you can do it in software, do it in software!

Clean successor of RISC.

Forms of Parallelism

- Pipelining
- Data parallelism (SIMD, Vector processing)
- Instruction level parallelism (ILP)
- Thread level parallelism (TLP)

Sequential Execution

- Process one instruction at a time
- No parallelism at all

Pipelined Execution

- Divide instructions into stages
- Parallel processing of independent stages
Parallel Execution

- Group independent operations
- Parallel processing of operations

Example: MIPS 74k

```c
void vec_sum(int *a, int *b, int *c, int n) {
    for (int i = 0; i < n; i++)
        (*c++) = (*a++) + (*b++);
}
```

Example: ST231

- Compiler generates 2 loops
  - first captures initial (n & 3) iterations
  - second is 4 times unrolled

VLIW vs. Superscalar
Architecture Classification

<table>
<thead>
<tr>
<th>Interchangeability</th>
<th>Independence of intrinsic</th>
<th>Independence of intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect use of register names</td>
<td>Exact description of all dependencies</td>
<td>Explicit declarations of both independent</td>
</tr>
<tr>
<td>Independent operations explicitly exposed</td>
<td>By the hardware's control unit</td>
<td>By the compiler and they are included in the program</td>
</tr>
<tr>
<td>Independent operations implicitly exposed</td>
<td>By the hardware's control unit</td>
<td>By the compiler and they are included in the program</td>
</tr>
<tr>
<td>Replacement of some operations scheduling capability</td>
<td>In the hardware's control unit</td>
<td>In the compiler</td>
</tr>
</tbody>
</table>

Role of the Compiler

- Compilers are important
  - Especially for VLIW architectures
  - Less for superscalar

- Extracting parallelism
  - Instruction scheduling
  - Predication
  - Speculation
  - Loop optimizations

VLIW vs. Superscalar

**Superscalar**
- Scheduling is done by hardware
- Sequential stream of scalar operations
- Allows for in-order and out-of-order execution
- Number of issued instructions is dynamically determined by a hardware dispatch unit
- Microarchitecture technique

**VLIW**
- Scheduling is done by software
- Sequential stream of parallel operations
- Only allows for in-order issue
- Number of issued instructions is statically determined by the compiler
- Architecture technique

The Role of the Compiler

- Compilers are important
  - Especially for VLIW architectures
  - Less for superscalar

- Extracting parallelism
  - Instruction scheduling
  - Predication
  - Speculation
  - Loop optimizations

Instruction Set

- Interface to the programmer/compiler
- Explicitly hide or expose architectural features

- Instruction encoding
  - Bundles/groups of operations
  - Legal combinations

- Binary compatibility
Architectural Features

- Execution model
  - Operation latencies
  - Computational resources
  - Semantics of parallel execution
- Exceptions/Interrupts
- Extensions
  - Predication
  - Speculation

Execution Model

- Many details are exposed to the programmer/compiler
  - Which operations are executed in parallel
  - How an operation is executed
  - When an operation is finished
  - Handling of hazards
- Complicates compilers
- Simplifies hardware

Semantics

- What is the value of r1?

```assembly
  mov $r1 = 1
  mov $r2 = 2;
  mov $r3 = 3;
  mov $r1 = $r2;;
```

- Is this valid?

```assembly
  {
    mov r1, 0
    if (r2 == 1) mov r1, 1
  }
```

Exceptions/Interrupts

- Exceptions may be raised in parallel
  - Precise vs. imprecise exceptions
  - Which instruction(s) caused the exception(s)
  - In which order are they processed
- Restarting execution
  - Which operations need to be reexecuted
Predication

- Conditionally nullify the effect of operations
- Full predication
  - All (almost all) operations can be predicated
- Partial predication
  - Only a few instructions can be predicated
  - Conditional move (cmov)
  - Select

Speculation

- Speculatively execute operations
  - Even if the calculation is useless
  - Even if the calculation may be incorrect
- May require compensation
  - Suppress exceptions
  - Undo incorrect calculations

Instruction Encoding

- Bridges between software and hardware
  - Programs are transformed to binary code
  - Hardware executes programs based on binary code
- Strong connection between architectural style and encoding
  - Encoding for RISC machines
  - CISC encoding techniques
  - Special techniques for VLIW

RISC Encoding

- RISC advocates simplicity and regularity
  - Encoding uses a fixed length
  - Few encoding formats
  - Reserved space for future extensions
- Simple decoding hardware
- Some code size overhead
Example: MIPS

- fixed width of 32 bit
- 3 encoding formats:

R-Format:

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rl</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

J-Format:

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rl</th>
<th>address/mediate</th>
</tr>
</thead>
</table>

Example: x86-32

- 7 addressing modes
- 5 different prefixes
- 4 displacement variants
- 4 immediate variants
- 1 – 17 bytes in size

CISC Encoding

- Variable length
  - Frequent instructions get short encoding
  - Infrequent instructions get longer encodings
  - Easier to add extensions
- Complex structure
  - Encode length of the instruction
  - Many instructions and instruction variants
  - More complex hardware

VLIW Encoding

- Inspired by RISC encoding
- Decompose a VLIW into fixed sized junks
  - Fixed encoding width for operations
  - Few encoding formats for operations
  - Variable encoding width for instructions
- Simple encoding scheme for bundles
Terminology

- Instruction/group
  - Independent operations that can be executed in parallel
- Bundle
  - Group of operations that are encoded in the same VLIW
  - Not necessarily independent

Example: Intel Itanium

- 41 bit syllable
- 41 bit operations (exactly one syllable)
- Bundle
  - 3 syllables/operations
  - 5 bits template and stop bit
- Instruction/group
  - Several bundles
  - Variable length

Terminology (2)

- Operation
  - Basic operation of the execution pipeline
  - Similar to RISC operations/instructions
- Syllable
  - Basic unit for the instruction encoding
  - Fixed bit width
  - Typically encodes one single operation

VLIW Encoding Schemes

- Uncompressed Encoding
- Fixed-overhead Encoding
- Distributed Encoding
- Template-based Encoding
Uncompressed Encoding

- Explicitly encode all operations
  - Including explicit NOPs if no useful operations could be found
- Allows for simple decoding
- Very bad code size
  - Negative effect on instruction cache

Saving NOPs

- Horizontal NOPs
  - Replace consecutive NOPs in a bundle
- Vertical NOPs
  - Replace consecutive bundles of NOPs
- Dynamically expanded during decoding

Fixed-overhead Encoding

- Variable length bundles
  - Based on horizontal NOPs
- Prepend a header to each bundle
  - Count of operations
  - Map operations to functional units
- Adopted by early architectures
  - e.g. Multiflow

Fixed-overhead Encoding (2)
**Distributed Encoding**

- Header information distributed
  - Either using a stop-bit or parallel-bit
  - Encoded with operations
- Reduces code size
- More complex decoding
  - Requires a search for the stop-bit
- Adopted by ST231

**Template-based Encoding**

- Similar to fixed-overhead encoding
  - Limits number of combinations
- Space efficient
- Low hardware overhead
- Adds complexity to compiler
- Adopted by Intel Itanium
Dispatching

- Assign operations to computational units
- Explicitly encoded in the simple scheme
  - 1:1 mapping of operations and functional units
- The mapping is lost for the other schemes

Dispatch (2)

- Unit identifiers within operations
- Explicit mapping
  - Fixed-overhead encoding
  - Within templates
- Positional encoding
  - Based on syllable ordering within instructions

Encoding Tricks

- Embedding large immediates
  - Special immediate operation
  - Used by some parallel operation
- Adopted by ST231

Encoding Tricks (2)

- Modifying the effect of parallel operations
  - Special test/modifier operations
- Adopted by Chili
Outlook

- Historical Perspective
- Architectural Structures
- Microarchitectural Design Issues
- Clustered Architectures
- Examples: Multiflow, Chili, ST231