Very Long Instruction Word Architecture and the ELI-512

Joseph A. Fisher
Yale University
ISCA 1983

Presenter: Alexander Eichhorn

25.11.2021
Executive Summary

• **Motivation**
  • Processors don’t make full use of independent instructions

• **Idea**
  • Have a Very Long Instruction Word with multiple instructions in one
  • Build a compiler to schedule independent instruction in parallel using *Trace Scheduling, Loop Unrolling, Memory Bank Prediction*
  • Build a hardware prototype: ELI-512

• **Results**
  • 10-30 speedup for ELI-512
Outline

• Problems & Main Idea of VLIW
• VLIW Compiler
  • Trace Scheduling
  • Memory Anti-Aliasing
  • Jump Mechanisms
  • Memory Bank Prediction
• ELI-512
• Current Use
• Strengths & Weaknesses
• Discussion
Problem

• Speed

• Parallel execution

• Speedup from parallelism never more than 2 or 3
Superscalar

instructions

1. $t0 = \text{zero} + 3$
2. $t0 = t0 - 4$
3. $t1 = t1 + a0$
4. $a0 = a0 + 1$

hardware

1. cycle
   - $t0 = \text{zero} + 3$
   - $t1 = t1 + a0$
2. cycle
   - $t0 = t0 - 4$
   - $a0 = a0 + 1$

independent instructions scheduled to run in same cycle at runtime
VLIW

Very Long Instruction Word

**previous instructions**

- addi $t0, $zero, 3
- addi $t0, $t0, -4
- add $t1, $t1, $a0
- addi $a0, $a0, 1

**VLIW compiled instructions**

1. instruction
- addi $t0, $zero, 3
- add $t1, $t1, $a0

2. instruction
- addi $t0, $t0, -4
- addi $a0, $a0, 1

independent instructions scheduled to run in same cycle at compile time
### VLIW

**Very Long Instruction Word**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lw $t3, 12($a0)</td>
<td></td>
</tr>
<tr>
<td>addi $t2, $t1, $t1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $t6, $t5, $t4</td>
<td></td>
<td>sw $t4, $s1, 0($sp)</td>
</tr>
<tr>
<td>lw $t3, 4($a0)</td>
<td></td>
<td>div $t4, $a1, $t1</td>
</tr>
<tr>
<td>move $a0, $s0</td>
<td></td>
<td>addi $sp, $sp, -8</td>
</tr>
</tbody>
</table>

**Execution Unit**

- Instruction: `lw $t3, 12($a0)`
- Instruction: `addi $t2, $t1, $t1`
- Instruction: `sub $t6, $t5, $t4`
- Instruction: `move $a0, $s0`
- Instruction: `lw $t3, 4($a0)`
- Instruction: `mult $t5, $t6, $t6`
- Instruction: `div $t4, $a1, $t1`
- Instruction: `add $sp, $sp, -8`
VLIW vs. Superscalar

**Advantage**

- Simpler hardware

**Disadvantage**

- Compiler needs to find independent instructions
- Code size increase in case there’s too much dependencies
Problem II: Compiler for VLIW

Simple Calculations

\[ x = (a + b) \ast (a - c) \]

\[ x_0 = a + b \]

\[ x_1 = a - c \]

\[ x = x_0 \ast x_1 \]

<table>
<thead>
<tr>
<th>( x_0 = a + b )</th>
<th>( x_1 = a - c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = x_0 \ast x_1 )</td>
<td>NOP</td>
</tr>
</tbody>
</table>
## Problem II: Compiler for VLIW

### Jumps

<table>
<thead>
<tr>
<th>fetch</th>
<th>decode</th>
<th>execute</th>
<th>writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult ...</td>
<td>xor ...</td>
<td>and ...</td>
<td>div ...</td>
</tr>
<tr>
<td>add ...</td>
<td>lw ...</td>
<td>lw ...</td>
<td>sw ...</td>
</tr>
<tr>
<td>slt ...</td>
<td>mult ...</td>
<td>or ...</td>
<td>addi ...</td>
</tr>
<tr>
<td>or ...</td>
<td>add ...</td>
<td>lw ...</td>
<td>beq ...</td>
</tr>
</tbody>
</table>
Problem II: Compiler for VLIW

Jumps

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>add ...</td>
<td>slt ...</td>
<td>or ...</td>
</tr>
<tr>
<td>lw ...</td>
<td>mult ...</td>
<td>add ...</td>
</tr>
<tr>
<td>lw ...</td>
<td>or ...</td>
<td>lw ...</td>
</tr>
<tr>
<td>sw ...</td>
<td>addi ...</td>
<td>beq ...</td>
</tr>
</tbody>
</table>

On misprediction of a branch, if we’re getting a 3 cycle penalty, we actually have to flush 3N (here 12) instructions
• Problems & Main Idea of VLIW
• VLIW Compiler
  • **Trace Scheduling**
  • Memory Anti-Aliasing
  • Jump Mechanisms
  • Memory Bank Prediction
• ELI-512
• Current Use
• Strengths & Weaknesses
• Discussion
• Trace Scheduling: A Technique for Global Microcode Compaction
  [J. Fisher, IEEE 1981]

• Divide code up into **basic blocks**

  • **Basic block** is a block of code that has no jumps in except at the beginning and no jumps out except at the end
Basic block is a block of code that has no jumps in except at the beginning and no jumps out except at the end.
Property of basic blocks

- **No outside influence**: registers/memory only changed by instructions inside block
- Easy to find independent instructions inside basic block
  - Rearrange independent instructions (traditionally)
  - Schedule independent instructions in same cycle (VLIW)
Compiler for VLIW

Trace Scheduling

\[ p_1 \quad 1 - p_1 \quad 1 - p_2 \quad p_2 \]

\[ 1 - p_3 \quad p_3 \quad 1 - p_4 \quad p_4 \]
1) determine most likely trace
2) combine trace into one basic block & reschedule code inside
3) add fix-up code to guarantee expected state when jumping out/in
4) recurse on not yet optimized code
Compiler for VLIW

Trace Scheduling

1) determine most likely trace
2) combine trace into one basic block & reschedule code inside
3) add fix-up code to guarantee expected state when jumping out/in
4) recurse on not yet optimized code
Compiler for VLIW

Trace Scheduling

1) determine most likely trace
2) combine trace into one basic block & reschedule code inside
3) add fix-up code to guarantee expected state when jumping out/in
4) recurse on not yet optimized code
Trace Scheduling

1) determine most likely trace
2) combine trace into one basic block & reschedule code inside
3) add fix-up code to guarantee expected state when jumping out/in
4) recurse on not yet optimized code
Compiler for VLIW

Trace Scheduling

1) determine most likely trace
2) combine trace into one basic block & reschedule code inside
3) add fix-up code to guarantee expected state when jumping out/in
4) recurse on not yet optimized code
Loops

• Supported by trace scheduling

• However: can be optimized using **loop unrolling**
  
  • by compiler instead of programmer
Compiler for VLIW

Trace Scheduling

loop body
Trace Scheduling

Compiler for VLIW

unrolled $k$ times
unrolled $k$ times $\rightarrow$ combined into one block to optimize
Compiler for VLIW

Trace Scheduling
Outline

• Problems & Main Idea of VLIW
• VLIW Compiler
  • Trace Scheduling
  • Memory Anti-Aliasing
  • Jump Mechanisms
  • Memory Bank Prediction
• ELI-512
• Current Use
• Strengths & Weaknesses
• Discussion
Compiler for VLIW

Memory Anti-Aliasing

\[ Z = A + X \]
\[ A = Y + W \]

can’t reschedule code lines
Memory Anti-Aliasing

\[ Z = A[\text{expr1}] + X \]
\[ A[\text{expr2}] = Y + W \]

• If \( A \) is an array, the compiler tries to solve the equation \( \text{expr1} = \text{expr2} \)

• Assume \( \text{expr1} \) and \( \text{expr2} \) are integers

• Use \textit{diophantine equation solver}

  • no solution = \( \text{expr1} \) and \( \text{expr2} \) are not the same

  \( \rightarrow \) can be rescheduled
Compiler for VLIW

Memory Anti-Aliasing

Example

```java
for (i = k to n) {
    Z = A[i*i] + X
    A[i+1] = Y + W

    ...
}
```

• Solve equation: \( i^2 = i + 1 \)

• There exist non-integer solutions: \( \frac{1}{2} \pm \frac{\sqrt{5}}{2} \)

• But no integer solutions exist

→ \( A[i*i] \) and \( A[i+1] \) never access the same element
Outline

- Problems & Main Idea of VLIW
- VLIW Compiler
  - Trace Scheduling
  - Memory Anti-Aliasing
  - **Jump Mechanisms**
    - Memory Bank Prediction
- ELI-512
- Current Use
- Strengths & Weaknesses
- Discussion
Jump Mechanisms

beq $t0, $t1, A  beq $t2, $t3, B  beq $t4, $t5, C  beq $t6, $t7, D

j A  j B  beq $t4, $t5, C  beq $t6, $t7, D

→ leads to ambiguity when executed
Jump Mechanisms

Possibilities

• Only allow 1 jump/branch instruction per cycle
• Multiple branch instructions with priority rules
Jump Mechanisms

Solution

• per cycle:
  • n independent tests
  • n+1 location to jump to
• Earlier tests have priority

COND (test1 label1)
    (test2 label2)
    ·
    ·
    ·
    (testn labeln)
    (TRUE label-fall-through)

C-style implementation

if (test1) goto label1
if (test2) goto label2
    ·
if (testn) goto labeln
goto label-fall-through
Jump Mechanisms

Compiler for VLIW

Encoding Size

- Placing n+1 full address candidates in instruction
  - Uses a lot of instruction bits

- Conform to certain regularities
# Jump Mechanisms

Compiler for VLIW

<table>
<thead>
<tr>
<th>Test Field</th>
<th>Condition</th>
<th>Potential Jump Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>test1</td>
<td>00 [NEXT INSTRUCTION ADDRESS]</td>
</tr>
<tr>
<td>1</td>
<td>test2</td>
<td>01 [NEXT INSTRUCTION ADDRESS]</td>
</tr>
<tr>
<td>2</td>
<td>test3</td>
<td>10 [NEXT INSTRUCTION ADDRESS]</td>
</tr>
<tr>
<td>3</td>
<td>TRUE</td>
<td>11 [NEXT INSTRUCTION ADDRESS]</td>
</tr>
</tbody>
</table>

\( n = 3 \)

**NEXT INSTRUCTION ADDRESS**: specified in each instruction
Jump Mechanisms

`n = 3`

**NEXT INSTRUCTION ADDRESS**: 00000011

<table>
<thead>
<tr>
<th>Test Field</th>
<th>Condition</th>
<th>Potential Jump Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>test1</td>
<td>00 00000011</td>
</tr>
<tr>
<td>1</td>
<td>test2</td>
<td>01 00000011</td>
</tr>
<tr>
<td>2</td>
<td>test3</td>
<td>10 00000011</td>
</tr>
<tr>
<td>3</td>
<td>TRUE</td>
<td>11 00000011</td>
</tr>
</tbody>
</table>
Compiler for VLIW

Jump Mechanisms

single branch with test1 can be encoded like this:

<table>
<thead>
<tr>
<th>Test Field</th>
<th>Condition</th>
<th>Potential Jump Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>test1</td>
<td>00 00000011</td>
</tr>
<tr>
<td>1</td>
<td>TRUE</td>
<td>01 00000011</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>10 00000011</td>
</tr>
<tr>
<td>3</td>
<td>TRUE</td>
<td>11 00000011</td>
</tr>
</tbody>
</table>
## Jump Mechanisms

**Compiler for VLIW**

**or jump to 10 11010111:**

<table>
<thead>
<tr>
<th>Test Field</th>
<th>Condition</th>
<th>Potential Jump Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FALSE</td>
<td>00 11010111</td>
</tr>
<tr>
<td>1</td>
<td>FALSE</td>
<td>01 11010111</td>
</tr>
<tr>
<td>2</td>
<td>TRUE</td>
<td>10 11010111</td>
</tr>
<tr>
<td>3</td>
<td>TRUE</td>
<td>11 11010111</td>
</tr>
</tbody>
</table>
Outline

• Problems & Main Idea of VLIW
• VLIW Compiler
  • Trace Scheduling
  • Memory Anti-Aliasing
  • Jump Mechanisms
  • **Memory Bank Prediction**
• ELI-512
• Current Use
• Strengths & Weaknesses
• Discussion
Compiler for VLIW

Memory Bank Prediction

Problem

• Multiple memory references in same cycle
  • Requires global arbitration system
    → latency increase
  • Possible memory bank conflict
    → freezes entire processor
Compiler for VLIW

Memory Bank Prediction

Solution

• Static code
  
  • Scalars always have known locations

  • Array values can be predicted by the same system that does anti-aliasing

• Loops

  • Often unrolling by a multiple of number of banks allows to predict banks

! Subscript of the array might have data-dependent starting point
Solution for data-dependent starting point: adding a pre-loop

```c
for i = k to n {
    arr[i] += 1
}

i = k
if (i >= n) goto FALLTHROUGH
arr[i] += 1
i++
if (i >= n) goto FALLTHROUGH
arr[i] += 1
i++
.
.
.
if (i < n) goto LOOP

LOOP:  
    if (i >= n) goto FALLTHROUGH
    arr[i] += 1
    i++
    if (i >= n) goto FALLTHROUGH
    arr[i] += 1
    i++
    ...
    if (i < n) goto LOOP

FALLTHROUGH:

PRELOOP:  
    if (i mod #BANKS == 0) goto LOOP
    arr[i] += 1
    i++
    if (i > n) goto FALLTHROUGH
    goto PRELOOP

LOOP:  
    if (i >= n) goto FALLTHROUGH
    arr[i] += 1
    i++
    if (i >= n) goto FALLTHROUGH
    arr[i] += 1
    i++
    ...
    if (i < n) goto LOOP

FALLTHROUGH:
```
Outline

• Problems & Main Idea of VLIW
• VLIW Compiler
  • Trace Scheduling
  • Memory Anti-Aliasing
  • Jump Mechanisms
  • Memory Bank Prediction
• ELI-512
  • Current Use
  • Strengths & Weaknesses
  • Discussion
ELI-512

- ELI = Enormously Longword Instructions
- 512 bit instruction word
- 16 clusters with an ALU and some storage
  - 8 M-clusters and 8 F-clusters
ELI-512

**M-clusters**
- Integer ALU
- Multiport integer register bank
- Local memory module
  - directly accessed when bank is known
  - Additionally 2 access ports to address memory globally (bank unknown) → slower

**F-clusters**
- Floating point ALU
- Multiport floating register bank
ELI-512
More about how the compiler has to optimize for these connections:

Parallel Processing: A Smart Compiler and a Dumb Machine
[J. Fisher et al., 1984]
Each instruction cycle:

- 16 ALU operations (8 restricted to 32-bit integer)
- 8 pipelined memory references
- 1 multiway conditional jump
Outline

• Problems & Main Idea of VLIW
• VLIW Compiler
  • Trace Scheduling
  • Memory Anti-Aliasing
  • Jump Mechanisms
  • Memory Bank Prediction
• ELI-512
• Current Use
  • Strengths & Weaknesses
  • Discussion
Current Technologies for Concurrency

Multicore

- Programmer has to manage concurrency
- No “micro-concurrency”
  - e.g. $x = (a + b) \times (a - c)$ can’t be efficiently calculated in parallel
- Much more efficient for completely independent programs
VLIW

Current Technologies for Concurrency

SIMD

• Programmer or compiler can manage concurrency

• Optional in CPU

• Obviously only usable for same instruction
VLIW Commercial Use

- **Intel Itanium Architecture** (IA-64) on EPIC design concept (1990)
  - Windows XP ported to IA-64
  - but mainly focussed on server market
  - discontinued 2010
- **AMD GPU**: Radeon HD 2900 with VLIW5 (2007)
  - Radeon HD 6900 with VLIW4
  - discontinued 2011
VLIW

Commercial Use

• **Xilinx Versal Chip (2019)**
  • AI engine with 6-way VLIW instructions
    • 2x scalar operations
    • 2x memory loads
    • 1x vector multiplication
    • 1x memory store
Outline

• Problems & Main Idea of VLIW
• VLIW Compiler
  • Trace Scheduling
  • Memory Anti-Aliasing
  • Jump Mechanisms
  • Memory Bank Prediction
• ELI-512
• Current Use
• Strengths & Weaknesses
• Discussion
Strengths

• No work for programmer to achieve concurrency
• No hardware scheduling
• A lot of solutions for different problems
• Compiler optimization can be used outside VLIW
Weaknesses

• **Compiler is computer specific**: instruction per cycle, latency of instructions, memory bank
  
  • recompilation for every change

• If one operation stalls, the whole processor stalls

• Inefficient **variable latency** operations
  
  • e.g. memory access with cache

• **Code size increase**

• No clear results / background of specified speedups not really clear

• ELI-512 not discussed in detail
Discussion

Why was VLIW never really commercially successful?
Discussion

For which today’s applications might VLIW be useful? or which emerging technologies?
Discussion

Adding an extra VLIW core to multicore processors today. Good idea? Challenges?