What Is VLIW?
- VLIW hardware is simple and straightforward,
- VLIW separately directs each functional unit

\[
\text{add r1,r2,r3} \\
\text{load r4,r5+4} \\
\text{mov r6,r2} \\
\text{mul r7,r8,r9}
\]

Historical Perspective: Microcoding, nanocoding (and RISC)
- A generation of high-performance, application-specific computers relied on horizontally microprogrammed computing engines.
- Aggressive (but tedious) hand programming at the microcode level provided performance well above sequential processors.

Horizontal Microcode and VLIW

Very Long Instruction Word (VLIW) Architectures
55:132/22C:160
High Performance Computer Architecture
Principles of VLIW Operation

- Statically scheduled ILP architecture.
- Wide instructions specify many independent simple operations.
- Multiple functional units execute all of the operations in an instruction concurrently, providing fine-grain parallelism within each instruction.
- Instructions directly control the hardware with no interpretation and minimal decoding.
- A powerful optimizing compiler is responsible for locating and extracting ILP from the program and for scheduling operations to exploit the available parallel resources.

The processor does not make any run-time control decisions below the program level.

Formal VLIW Models

- Josh Fisher proposed the first VLIW machine at Yale (1983).
- Fisher’s Trace Scheduling algorithm for microcode compaction could exploit more ILP than any existing processor could provide.
- The ELI-512 was to provide massive resources to a single instruction stream:
  - 16 processing clusters - multiple functional units/cluster.
  - Partial crossbar interconnect.
  - Multiple memory banks.
  - Attached processor – no I/O, no operating system.
- Later VLIW models became increasingly more regular:
  - Compiler complexity was a greater issue than originally envisioned.

Ideal Models for VLIW Machines

- Almost all VLIW research has been based upon an ideal processor model.
- This is primarily motivated by compiler algorithm developers to simplify scheduling algorithms and compiler data structures.
  - This model includes:
    - Multiple universal functional units
    - Single-cycle global register file
    - Single-cycle execution
    - Unrestricted, multi-ported memory
    - Multi-way branching
    - Sometimes:
      - Unlimited resources (functional units, registers, etc.)

VLIW Execution Characteristics

- Basic VLIW architectures are a generalized form of horizontally microprogrammed machines.
Unresolved design issues
- The best functional unit mix
- Register file and interconnect topology
- Memory system design
- Best instruction format

Many questions could be answered through experimental research
- Difficult - needs effective retargetable compilers

Compatibility issues still limit interest in general-purpose VLIW technology

However, VLIW may be the only way to build 8-16 operation/cycle machines.

---

Scheduling for Fine-Grain Parallelism

- The program is translated into primitive RISC-style (three address) operations
- Dataflow analysis is used to derive an operation precedence graph from a portion of the original program
- Operations which are independent can be scheduled to execute concurrently contingent upon the availability of resources
- The compiler manipulates the precedence graph through a variety of semantic-preserving transformations to expose additional parallelism

---

Example

Original Program:
\[
e = (a + b) \cdot (c + d)
\]
b++;    

3-Address Code:
\[
\begin{align*}
r1 &= a + b \\
e &= r1 \cdot r2 \\
b &= b + 1
\end{align*}
\]

Dependency Graph:

VLW Instructions:
- add a,b,r1
- add c,d,r2
- add b,1,e
- mul r1,r2,e
- nop

---

Realistic VLIW Datapath

- Multi-Ported Register File
- Instruction Memory
- Sequencer
- Condition Codes

Instructions:
- FAdd (1 cycle)
- FMul 4 cyc pipe
- FMul 4 cyc unpipe
- FDiv 16 cycle

No Bypass!!
No Stall!!
**VLIW List Scheduling**

- Assign Priorities
- Compute Data Ready List - all operations whose predecessors have been scheduled.
- Select from DRL in priority order while checking resource constraints
- Add newly ready operations to DRL and repeat for next instruction

**Enabling Technologies for VLIW**

- VLIW Architectures achieve high performance through the combination of a number of key enabling hardware and software technologies.
  - Optimizing Schedulers (compilers)
  - Static Branch Prediction
  - Symbolic Memory Disambiguation
  - Predicated Execution
  - (Software) Speculative Execution
  - Program Compression

**Strengths of VLIW Technology**

- Parallelism can be exploited at the instruction level
  - Available in both vectorizable and sequential programs.
- Hardware is regular and straightforward
  - Most hardware is in the datapath performing useful computations.
  - Instruction issue costs scale approximately linearly
- Architecture is "Compiler Friendly"
  - Implementation is completely exposed - 0 layer of interpretation
  - Compile time information is easily propagated to run time.
- Exceptions and interrupts are easily managed
- Run-time behavior is highly predictable
  - Allows real-time applications.
  - Greater potential for code optimization.

**Weaknesses of VLIW Technology**

- No object code compatibility between generations
- Program size is large (explicit NOPs)
  - Multiflow machines predated "dynamic memory compression" by encoding NOPs in the instruction memory
- Compilers are extremely complex
  - Assembly code is almost impossible
- Philosophically incompatible with caching techniques
- VLIW memory systems can be very complex
  - Simple memory systems may provide very low performance
  - Program controlled multi-layer, multi-banked memory
- Parallelism is underutilized for some algorithms.
VLIW vs. Superscalar [Bob Rau, HP]

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Superscalar</th>
<th>VLIW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple instructions/cycle</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Multiple operations/Instruction</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Instruction stream parsing</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Run-time analysis of register dependencies</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Run-time analysis of memory dependencies</td>
<td>maybe</td>
<td>occasionally</td>
</tr>
<tr>
<td>Runtime instruction reordering</td>
<td>maybe</td>
<td>no</td>
</tr>
<tr>
<td>Runtime register allocation</td>
<td>maybe</td>
<td>maybe</td>
</tr>
</tbody>
</table>

Real VLIW Machines

- VLIW Minisupercomputers/Superminicomputers:
  - Multiflow TRACE 7/300, 14/300, 28/300 [Josh Fisher]
  - Multiflow TRACE 1500 [Bob Colwell]
  - Cydrome Cydra 5 [Bob Rau]
  - IBM Yorktown VLIW Computer (research machine)
- Single-Chip VLIW Processors:
  - Intel iWarp, Philip’s LIFE Chips (research)
- Single-Chip VLIW Media (through-put) Processors:
  - Trimedia, Chromatic, Micro-Unity
- DSP Processors (TI TMS320C6x)
  - Intel/HP EPIC IA-64 (Explicitly Parallel Instruction Comp.)
  - Transmeta Crusoe (x86 on VLIW?)
- Sun MAJC (Microarchitecture for Java Computing)

Why VLIW Now?

- Nonscalability of Superscalar Processor
  - ILP and complexity
- Better compilation technology

Performance Obstacles of Superscalars

- Branches
  - branch prediction helps, but penalty is still significant
  - limits scope of dynamic and static ILP analysis + code motion
- Memory Load Latency
  - CPU speed increases at 60% per year
  - memory speed increases only 5% per year
- Memory Dependence
  - disambiguation is hard, both in hardware and software
- Sequential Execution Semantics ISAs
  - total ordering of all the instructions
  - implicit inter-instruction dependences

Very expensive to implement wide dynamic superscalars
**Intel/HP EPIC/IA-64 Architecture**

- **EPIC (Explicitly Parallel Instruction Computing)**
  - An ISA philosophy/approach
  - e.g. CISC, RISC, VLIW
  - Very closely related to but not the same as VLIW
- **IA-64**
  - An ISA definition
  - e.g. IA-32 (was called x86), PA-RISC
  - Intel’s new 64-bit ISA
  - An EPIC type ISA
- **Itanium (was code named Merced)**
  - A processor implementation of an ISA
  - e.g. P6, PA8500
  - The first implementation of the IA-64 ISA

**IA-64 EPIC vs. Classic VLIW**

- **Similarities:**
  - Compiler generated wide instructions
  - Static detection of dependencies
  - ILP encoded in the binary (a group)
  - Large number of architected registers
- **Differences:**
  - Instructions in a bundle can have dependencies
  - Hardware interlock between dependent instructions
  - Accommodates varying number of functional units and latencies
  - Allows dynamic scheduling and functional unit binding
    - Static scheduling are “suggestive” rather than absolute
    - Code compatibility across generations
      - but software won’t run at top speed until it is recompiled so “shrink-wrap binary” might need to include multiple builds