

VLIW Processors

VLIW (“very long instruction word”) **processors**

- instructions are scheduled by the compiler
- a fixed number of operations are formatted as one big instruction (called a **bundle**)
 - usually **LIW** (3 operations) today
 - change in the instruction set architecture, i.e., 1 program counter points to 1 bundle (not 1 operation)
- want operations in a bundle to issue in parallel
 - fixed format so could decode operations in parallel
 - enough FUs for types of operations that can issue in parallel
 - pipelined FUs

VLIW Processors

Roots of modern VLIW machines

Multiflow & Cydra 5 (8 to 16 operations) in the 1980's

Today's VLIW machines

Itanium (3 operations)

Transmeta Crusoe (4 operations)

embedded processors

VLIW Processors

Goal of the VLIW design: reduce hardware complexity

- less design time
- shorter cycle time
- better performance
- reduced power consumption

How VLIW designs reduce hardware complexity (in theory)

- less multiple-issue hardware
 - no dependence checking for instructions within a bundle
 - can be fewer paths between instruction issue slots & FUs
- simpler instruction dispatch
 - no out-of-order execution, no instruction grouping
- no structural hazard checking logic

Compiler figures all this out.

VLIW Processors

Compiler support to increase ILP

- compiler creates each VLIW word
- greater need for good code scheduling than with in-order issue superscalars
 - instruction doesn't issue if 1 operation can't

VLIW Processors

More **compiler support** to increase ILP

- detects hazards & hides latencies
 - structural hazards
 - no 2 operations to the same functional unit
 - no 2 operations to the same memory bank
 - data hazards
 - no data hazards among instructions in a bundle
 - control hazards
 - predicated execution
 - static branch prediction
 - hiding latencies
 - data prefetching
 - hoisting loads above stores

Compiler Support for Increasing ILP

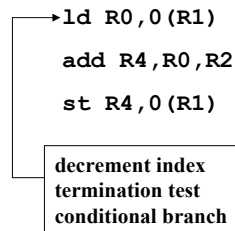
Compiler optimizations that increase ILP

- loop unrolling
- aggressive inlining: function becomes part of the caller code
- software pipelining: schedules instructions from different iterations together
- trace scheduling & superblocks: schedule beyond basic block boundaries

Compiler Support for Increasing ILP

Software pipelining

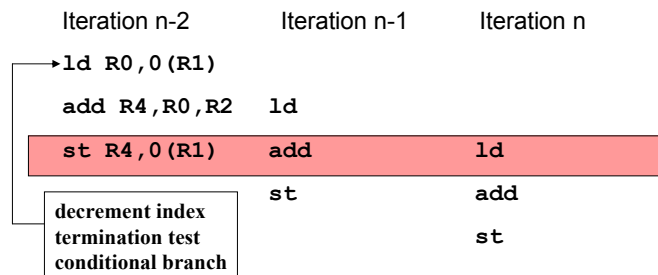
- a simple example



Compiler Support for Increasing ILP

Software pipelining

- schedules instructions from different iterations together



Compiler Support for Increasing ILP

Software pipelining

- handling memory accesses

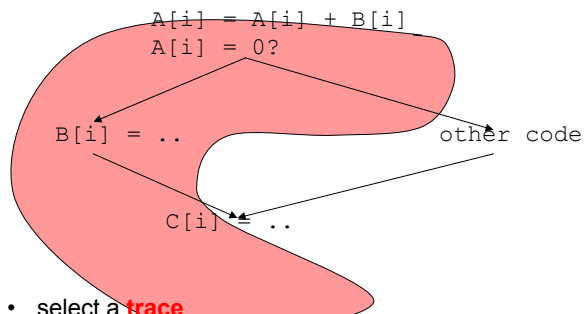
<code>st R4, 16(R1)</code>	stores into mem[i]
<code>add R4, R0, R2</code>	adds to mem[i-1]
<code>ld R0, 0(R1)</code>	loads from mem[i-2]

- performance advantages: increasing ILP
- performance disadvantages: still executing loop control instructions

Compiler Support for Increasing ILP

Global scheduling (trace scheduling & superblocks)

- schedule beyond basic block boundaries



- select a **trace**
- compact instructions within it
- compensate for crossing basic block boundaries

Compiler Support for Increasing ILP

Compiler optimizations that increase ILP

- unroll the trace
- **trace scheduling:**
 - trace entrances & exits at each iteration
 - more difficulty & more compensation code than
- **superblocks:** trace exits at each iteration
 - one trace entrance & multiple exits (each iteration)
 - more difficulty & more compensation code than
- advantages depend on path frequencies, empty instruction slots, whether moved instruction is the beginning of a critical path, amount of compensation code on non-trace path

IA-64 EPIC

Explicitly **P**arallel **I**nstruction **C**omputing, aka VLIW
IA-64 architecture, Itanium implementation

Bundle of instructions

- 128 bit bundles
- 3 instructions/bundle
- 2 bundles can be issued at once
 - if issue one, get another

IA-64 EPIC

Registers

- 128 integer & FP registers
 - implications for architecture?
- 128 additional registers for loop unrolling & similar optimizations
 - implications for hardware?
- miscellaneous other registers
- implications for performance?
 - +
 - +
 -
 -

IA-64 EPIC

Full predicated execution

- supported by 64 one-bit predicate registers
 - instructions can set 2 at once (comparison result & complement)
- example

```
    cmp.eq r1, r2, p1, p2
    (p1) sub 59, r10, r11
    (p2) add r5, r6, r7
```

IA-64 EPIC

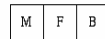
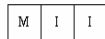
Full predicated execution

- implications for architecture?
- implications for the hardware?
- implications for exploiting ILP?

IA-64 EPIC

Template in each bundle that indicates:

- type of operation for each instruction
- instruction order in bundle
- examples (2 of 24)



- M: load & manipulate the address (e.g., increment an index)
- I: integer ALU op
- F: FP op
- B: transfer of control
- other, e.g., *stop* (see below)
- restrictions on which instructions can be in which slots
 - schedule code for functional unit availability (i.e., template types) & latencies

IA-64 EPIC

Template, cont'd.

- a stop bit that delineates the instructions that can execute in parallel
 - all instructions before a stop have no data dependences
- implications for hardware:
 - simpler issue logic, no instruction slotting, no out-of-order issue
 - potentially fewer paths between issue slots & functional units
 - potentially no structural hazard checks
 - hardware not have to determine intra-bundle data dependences

IA-64 EPIC

Branch support

- full predicated execution
- hierarchy of branch prediction structures in different pipeline stages
 - 4-target BTB for repeatedly executed taken branches
 - an instruction puts a specific target in it (i.e., the BTB is exposed to the architecture)
 - larger back-up BTB
 - correlated branch prediction for hard-to-predict branches
 - instruction hint that branches that are statically easy-to-predict should **not** be placed in it
 - private history registers, 4 history bits, shared PHTs
 - separate structure for multi-way branches
- branch prediction instruction for target forecasting
- branch prediction instruction for storing a prediction

IA-64 EPIC

ISA & microarchitecture seem complicated (some features of an out-of-order processors)

- not all instructions in a bundle need stall if one stalls (a scoreboard keeps track of produced values that will be source operands for stalled instructions)
- branch prediction hierarchy
- dynamically sized register stack, aka register windows
 - special hardware for register window overflow detection
 - special instructions for saving & restoring the register stack
- register remapping to support rotating registers on the “register stack” which aid in software pipelining
- array address post-increment & loop control

IA-64 EPIC

More complication

- don't want to store speculative values to memory
 - special instructions check integer register poison bits to detect whether value is speculative (for nonspeculative code or exceptions)
 - OS can override the ban on storing (e.g., for a context switch)
 - different mechanism for speculative floating point values
- backwards compatibility
 - x86 (IA-32)
 - PA-RISC compatible memory model (segments)

Trimedia TM32

Designed for the embedded market

Classic VLIW

- no hazard detection in hardware
 - nops “guarantee” that dependences are followed
- instructions decompressed on fetching

Superscalars vs. VLIW

Superscalar has more complex hardware for instruction scheduling

- instruction slotting or out-of-order hardware
- more paths or more complicated paths between instruction issue structure & functional units
- dependence checking logic between parallel instructions
- functional unit hazard checking
- possible consequences:
 - slower cycle times
 - more chip real estate
 - more power consumption

Superscalars vs. VLIW

VLIW has more functional units if supports full predication

- paths between instruction issue structure & more functional units
- possible consequences:
 - slower cycle times
 - more chip real estate
 - more power consumption

Superscalars vs. VLIW

VLIW has larger code size

- estimates of IA-64 code of up to 2X - 4X over x86
 - 128b holds 4 (not 3) instructions on a RISC superscalar
 - sometimes nops if don't have an instruction of the correct type
 - branch targets must be at the beginning of a bundle
 - predicated execution to avoid branches
 - extra, special instructions
 - check for exceptions
 - check for improper load hoisting (memory aliases)
 - allocate register windows for local variables
 - branch prediction
- consequences:

Superscalars vs. VLIW

VLIW requires a more complex compiler

- consequence: more design effort or poor quality code if good optimizations aren't implemented

Superscalars can more efficiently execute pipeline-dependent code

- consequence: don't have to recompile if change the implementation

What else?