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 (which Intel calls a **bundle**) which are fetched together
    - a change in the instruction set architecture, i.e., 1 program counter points to 1 bundle (not 1 operation)
    - usually **LIW** (3 operations) today
- want operations in a bundle to issue in parallel

VLIW Processors

Goal of the VLIW design: reduce hardware complexity

- less hardware design & test time
- shorter cycle time
- put the complexity in the compiler which can analyze the whole program

⟶ better performance

⟶ reduced power consumption

How VLIW designs reduce hardware complexity

- 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
- simpler structural hazard-checking logic

VLIW Processors

Compiler support to increase ILP

- compiler creates each VLIW word
- key role in identifying ILP
- need for good code scheduling & ILP-creating optimizations is greater than with in-order-issue superscalars
- VLIW instruction doesn't issue if one of its operations can't issue

---

VLIW Processors

More **compiler support** to increase ILP

- detects structural hazards
  - no 2 operations to the same functional unit
  - no 2 operations to the same memory bank
- detects data hazards
  - no data hazards among instructions in a bundle
- detects control hazards
  - predicated execution
  - static branch prediction
- hides latencies
  - data prefetching for arrays
  - hoisting loads above stores
  - hoisting a basic block above a branch (speculation)

VLIW Processors

Compiler optimizations that increase ILP

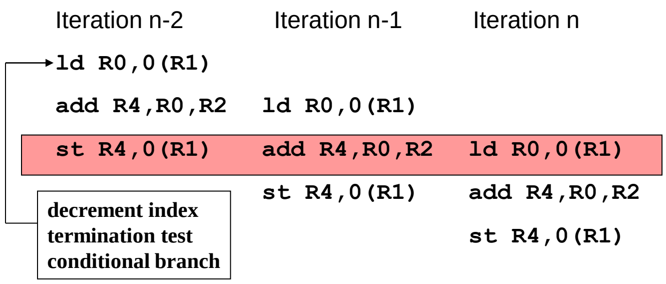
- loop unrolling
- aggressive inlining: function becomes part of the caller code
- software pipelining: schedules instructions from different iterations together
- trace scheduling & superblocks: schedule beyond basic block boundaries

---

VLIW Processors

Compiler optimizations that increase ILP

- **software pipelining**: schedules instructions from different iterations together



Iteration n-2	Iteration n-1	Iteration n
`ld R0,0(R1)`		
`add R4,R0,R2`	`ld R0,0(R1)`	
`st R4,0(R1)`	`add R4,R0,R2`	`ld R0,0(R1)`
	`st R4,0(R1)`	`add R4,R0,R2`
		`st R4,0(R1)`

decrement index
termination test
conditional branch

VLIW Processors

Compiler optimizations that increase ILP

- **software pipelining**: memory accesses

`st R0, 16(R1)`	stores into mem[i]	all in 1 cycle
`add R4, R0, R2`	computes on mem[i-1]	
`ld R4, 0(R1)`	loads from mem[i-2]	

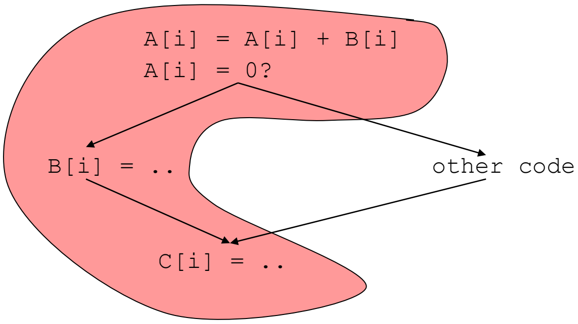
- performance advantages: increasing ILP
- performance disadvantages: still executing loop control instructions

---

VLIW Processors

Compiler optimizations that increase ILP

- **global scheduling** (**trace scheduling**, **superblocks**): schedule beyond basic block boundaries
- goal: to create longer path of sequential instructions



- example: trace scheduling (select a **trace**; then schedule its instructions)

VLIW Processors

Roots of modern VLIW machines

- to reduce hardware complxity: Multiflow & Cydra 5 (1980s)
  - 8 to 16 operations

Today's VLIW machines

- for performance: Intel Itanium (3 operations, 2000)
- for the low power & small size, the embedded market: Trimedia TM32 (5 operations)

IA-64 EPIC

Explicitly **P**arallel **I**nstruction **C**omputing, aka VLIW

1.67 GHz Itanium 2 implementation, IA-64 architecture

Bundle of instructions

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

IA-64 EPIC

Template in each bundle that indicates:

- types of operations that can be executed together (without hazards)
- instruction order in a bundle
- examples (2 of 24)

M	I	I

M	F	B

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

---

IA-64 EPIC

Template, cont'd.

- a stop bit that delineates which instructions can execute in parallel
  - all instructions before a stop have no data dependences

- implications for hardware:
  - simpler issue logic, no out-of-order issue
  - fewer paths between issue slots & functional units
  - simpler structural hazard checks
  - hardware not have to determine intra-bundle data dependences

IA-64 EPIC

Registers

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

- implications for hardware?

- implications for performance?
  +

  +

  -

  -

IA-64 EPIC

Full predicated execution

- supported by 64 one-bit predicate registers
  - instructions can set 2 at once (result of the comparison & its 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

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 easy-to-predict statically should ***not*** be placed in it
    - 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

Some more hardware complication

- 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)
- 128 registers can be used as a dynamically sized register stack, aka register windows
  - special hardware for register window overflow detection
  - special instructions for saving & restoring the register stack
- 128 registers use register remapping to increment register number (modulo the number of registers) to aid in software pipelining
- array address post-increment & loop control

IA-64 EPIC

Still more complication

- speculative values cannot be stored to memory
  - special instructions check integer register poison bits to detect whether value is speculative
  - 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
  - small, non-general, fixed code, often hand-optimized
- 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 structural hazard checking
- 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 on the register stack for local variables
        - branch prediction
- consequences:
    - increase in instruction bandwidth requirements
    - Increase instruction cache footprint & decrease I-cache effectiveness

---

Superscalars vs. VLIW

VLIW *requires* a more complex compiler

- consequence: 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?