18-547 Lecture 22
*ILP + Vector*
Instructor: Prof. Falsafi

Electrical & Computer Engineering
Carnegie Mellon University

Slides developed by Profs. Hill, Wood, Sohi, and Smith of
University of Wisconsin — Madison

Announcements

Homework 6 due next week
Guest lectures next week:
Monday, Prof. Marculescu, Power-Aware Microarchitecture
Wednesday, Prof. Nowatzyk, Multiprocessor Architecture

Last week: I/O & Interconnects

Final exam: Fri. December 14 5:30pm-8:30pm, PH 125C
Case Study: Alpha 21164

4-way superscalar - 2 memory, 2 ALU, 2 FP
issue in-order
interrupts RPL+bypasses; FP imprecise
1-deep branch speculation
direct-mapped 8 KB instruction and data caches
on-chip L2-cache
Case Study: Intel P6

CISC to ROP conversion
20 unified reservation stations
up to 3 simple instr decode/cycle; only 1 complex
ROB holds values
up to 5 ROPs issue per cycle; 3 ROPs commit per cycle
max 1 load and 1 store per cycle
CPU and L2-cache on MCM
512 entry, 4-way associative BTB
Case Study: Intel P6

12-stage pipeline, high branch penalty
First implementation of Yeh&Patt predictor
Superscalar Trends

Could continue with wider issue, larger reorder buffer

Increases IPC, but ignores clock cycle

Clock is a function of logic + wire delays

Larger windows, larger issue width

- more associative searches & tag matches in reservation stations
  => more logic per clock + longer paths per clock

In future, wire delays will dominate (see paper #5 by Yu)

- bypass paths already critical

Superscalar Trends

Wire delays don’t change with smaller feature sizes

1. smaller faster transistors
2. larger dies

=> longer wires on-chip will take multiple clocks
So on-chip communication, not computation will dominate
Case Study: Alpha 21264

6-way superscalar – 4 ALU + memory, 2 FP
issue out-of-order, retire in-order (precise)

- 2-way cluster, 1-cycle bypass
- 7-deep branch speculation
- Hybrid 1K local + 4K global (PAd & GA\text{g})
- 2-way 64 KB instruction and data caches
- Wave-pipelined
- Line & way prediction to reduce latency
- off-chip L2-cache
Clustering FUs

Reservation Stations

Reg. File0

Exec

Exec

CP delay

Reg. File1

Exec

Exec

VLIW: Software ILP

Very long instruction word
Provide a long instruction word with one operation per FU
Instruction latencies are fixed
Compiler packs independent instructions into VLIW
- compiler schedules all hardware resources
Entire long word issues as a “unit”
Result:

ILP with simple hardware, simple control, fast clock
VLIW: Scheduling

Code scheduling in software only
Loop unrolling, software pipelining, trace scheduling

Architectural support
- deferred interrupts
  - enhances scheduling opportunity
- predicated execution (conditional moves)
  - less need for hardware prediction
  - speculative loads + check instructions
- more registers
  - renaming not necessary?

Predicated or Conditional Execution

Instructions accompanied by condition registers!
- Condition true => instruction is executed
- Condition false => instruction annulled
- E.g., bnez R1, Skip
  
  
  mov R2, R3
Change to:
  
  cmove R2, R3, R1

Better flow of fetch and decode
May lead to lots of nops!
**VLIW: History**

Floating point systems, array processors
- very successful in 70’s
- all latencies fixed
- fast, single-level memory

Multiflow
- Josh Fisher (now at HP)
- 1980’s minicomputer

Cydrome
- Bob Rau (now at HP)
- 1980’s minicomputer

Tera
- Burton Smith, 1980’s-1990’s supercomputer, keeps failing?

Intel IA-64
- co-developed by HP
- latest news => compiler works well for regular applications!
- key design philosophy => simplify hardware
- in practice => more complex than anything they have built

Transmeta
- Dave Ditzel from the original RISC
- goal: better perf/transistor for x86 code
- technology: dynamic binary rewriting
- “software decode” to VLIW RISC ISA
- hard to beat x86 market (Intel’s margins are low)

Texas Instrument DSP processors => current success story
VLIW Problems

Same as those with static scheduling
Latencies are not fixed!
Caches are here to stay
Need mechanisms to deal with latency dynamically

Case Study: Itanium

6-way issue (2 VLIW “bundles” of 3 instructions)
Issue in-order, 10-stage pipeline
Predicated execution using condition registers
  • Compiler guarantees independence
  • Load speculation for memory operations
Hierarchy of branch predictors including two-level
  • Compiler branch hints
Register stack mapping
16K L1 I and D, 96K L2, 4M L3 caches
Other ILP Approaches: Vectors

A vector is a one-dimensional array of numbers
Many scientific and commercial programs use vectors
Numeric computations using floating-point
for (i = 1; i<=64; i++)
    a[i] = b[i] + c[i]
Integer computation on image pixels
Vector instructions perform an operation on each element
    addv    a, b, c
Why Vectors?

Vector instructions allow deeper pipelines
- no intra-vector interlocks
- no intra-vector hazards
- inner loop control hazards eliminated
- need not issue multiple instructions
- vectors can present memory access pattern to hardware

Want deeper pipelines but
- interlock logic is hard to divide into more stages
- bubbles due to data hazard increase
- hard to issue multiple instructions per cycle
- fetch&issue bottleneck (Flynn bottleneck)

Vector Architectures

Vector-Register Machines
- load/store architectures
- vector operations use vector register except ld/st
- register ports cheaper than memory ports
- optimized for small vectors
Vector Architectures (Cont.)

Memory-Memory vector machines

- all vectors reside in memory
- long startup latency
- memory ports expensive
- optimized for long vectors

Fact: most vectors are short

- early machines were memory-memory
- TI ASC, CDC STAR-100
- modern vector machines use vector-registers

DLXV Architecture

Strongly based on CRAY-1

Extend DLX with vector instructions

- original DLX is a scalar unit
- 32 integer and 32 FP registers

Eight vector registers (V0-V7)

- 64 double-precision FP each (4K bytes total)

Five vector functional units

- FP+, FP*, FP/, integer and logical
- fully-pipeline with 2-20 stages

Vector ld/st units

- fully-pipelined with 10-50 stages
DLXV Instructions

Vector-vector instructions
- operate on two vectors
- produce a third vector
- addv v1, v2, v3

Vector-scalar instructions
- operate on one vector and one scalar
- addv v1, f0, v3

Vector ld/st instructions
- ld/st a vector from memory into a vector register
- operates on contiguous addresses
- lv [r1], v1 ; v[l] = M[r1+l]
- sv v1, [r1] ; M[r1+l] = v[l]

DLXV Load/Store

ld/st vector with stride
- vectors are not always contiguous in memory
- add non-unit stride on each access
- lvws [r1,r2], v1 ; v[l] = M[r1+l*r2]
- svws v1, [r1, r2] ; M[r1+l*r2] = v[l]

ld/st indexed
- indirect accesses through an index vector
- lvws [r1,v2], v1 ; v[l] = M[r1+v2[l]]
- svws v1, [r1, v2] ; M[r1+v2[l]] = v[l]
Vector Code Example

DAXPY: double-precision a * x + y
for (i=1; i<=64;i++)
    y[i] = a*x[i]+y[i]
VLR 64
ld [a], f0
lv [rx], v1
multv v1, f0, v2
lv [ry], v3
addv v2, v3, v4
sv v4, [ry]

Not all vectors are 64 elements long

Vector length register (VLR)
  □ controls length of vector operations
  □ 0 < VLR < MVL = 64

Strip mining
  □ a loop of vector code
  □ each iteration implements the code for 64 elements
Other Vector Operations

Use masked vector register for vectorizing if statements
- a mask specifies which vector elements are operated on
- can set the mask using logical compares (sltsv v1, f0)

Vector chaining
- bypassing vector values between functional units
- multv --, --, v1 and addv --, v1, -- executing simultaneously

Scatter/gather
- used for sparse arrays/matrices
- using an index vector, gather elements into a vector register
- operate on the vector
- put the vector back

Vector Memory

What kind of memory hierarchy would you use for vectors?

Compiler techniques
Final word: make the scalar unit fast!
Remember Amdahl’s Law
CRAY-1 was the fastest scalar computer