CS252 Graduate Computer Architecture
Fall 2015
Lecture 10: VLIW Architectures

Krste Asanovic
krste@eecs.berkeley.edu
http://inst.eecs.berkeley.edu/~cs252/fa15
Last Time in Lecture 9

Vector supercomputers
- Vector register versus vector memory
- Scaling performance with lanes
- Stripmining
- Chaining
- Masking
- Scatter/Gather
Sequential ISA Bottleneck

Sequential source code

a = foo(b);
for (i=0, i<

Superscalar compiler
Find independent Schedule operations

Superscalar processor
Check instruction dependencies Schedule execution

Sequential machine code
VLIW: Very Long Instruction Word

- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no cross-operation RAW check
  - No data use before data ready => no data interlocks
Early VLIW Machines

- **FPS AP120B (1976)**
  - scientific attached array processor
  - first commercial wide instruction machine
  - hand-coded vector math libraries using software pipelining and loop unrolling

- **Multiflow Trace (1987)**
  - commercialization of ideas from Fisher’s Yale group including “trace scheduling”
  - available in configurations with 7, 14, or 28 operations/instruction
  - 28 operations packed into a 1024-bit instruction word

- **Cydrome Cydra-5 (1987)**
  - 7 operations encoded in 256-bit instruction word
  - rotating register file
VLIW Compiler Responsibilities

- Schedule operations to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedule to avoid data hazards (no interlocks)
  – Typically separates operations with explicit NOPs
How many FP ops/cycle?

1 fadd / 8 cycles = 0.125
Loop Unrolling

for (i=0; i<N; i++)

Unroll inner loop to perform 4 iterations at once

for (i=0; i<N; i+=4)
{
}

Need to handle values of N that are not multiples of unrolling factor with final cleanup loop
Scheduling Loop Unrolled Code

Unroll 4 ways

loop: fld f1, 0(x1)
  fld f2, 8(x1)
  fld f3, 16(x1)
  fld f4, 24(x1)
  add x1, 32
  fadd f5, f0, f1
  fadd f6, f0, f2
  fadd f7, f0, f3
  fadd f8, f0, f4
  fsd f5, 0(x2)
  fsd f6, 8(x2)
  fsd f7, 16(x2)
  fsd f8, 24(x2)
  add x2, 32
  bne x1, x3, loop

Schedule

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36
Software Pipelining

Unroll 4 ways first

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       add x2, 32
       fsd f8, -8(x2)
       bne x1, x3, loop
```

How many FLOPS/cycle?
4 fadds / 4 cycles = 1

```
        Int1 | Int2 | M1 | M2 | FP+ | FPx
------|------|----|----|-----|-----
 prolog: |      |    |    |     |     |
 loop: |      |    |    |     |     |
 epilog: |      |    |    |     |     |
```
Software Pipelining vs. Loop Unrolling

Software pipelining pays startup/wind-down costs only once per loop, not once per iteration
What if there are no loops?

- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks

Basic block
Trace Scheduling [Fisher, Ellis]

- Pick string of basic blocks, a trace, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole "trace" at once
- Add fixup code to cope with branches jumping out of trace
Problems with “Classic” VLIW

- Object-code compatibility
  - have to recompile all code for every machine, even for two machines in same generation

- Object code size
  - instruction padding wastes instruction memory/cache
  - loop unrolling/software pipelining replicates code

- Scheduling variable latency memory operations
  - caches and/or memory bank conflicts impose statically unpredictable variability

- Knowing branch probabilities
  - Profiling requires an significant extra step in build process

- Scheduling for statically unpredictable branches
  - optimal schedule varies with branch path
VLIW Instruction Encoding

- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - used in Multiflow Trace
    - introduces instruction addressing challenge
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64
  - Provide a single-op VLIW instruction
    - Cydra-5 UniOp instructions
Intel Itanium, EPIC IA-64

- EPIC is the style of architecture (cf. CISC, RISC)
  - Explicitly Parallel Instruction Computing (really just VLIW)
- IA-64 is Intel’s chosen ISA (cf. x86, MIPS)
  - IA-64 = Intel Architecture 64-bit
  - An object-code-compatible VLIW
- Merced was first Itanium implementation (cf. 8086)
  - First customer shipment expected 1997 (actually 2001)
  - McKinley, second implementation shipped in 2002
  - Recent version, Poulson, eight cores, 32nm, announced 2011
Eight Core Itanium “Poulson” [Intel 2011]

- 8 cores
- 1-cycle 16KB L1 I&D caches
- 9-cycle 512KB L2 I-cache
- 8-cycle 256KB L2 D-cache
- 32 MB shared L3 cache
- 544mm² in 32nm CMOS
- Over 3 billion transistors
- Cores are 2-way multithreaded
- 6 instruction/cycle fetch
  - Two 128-bit bundles
- Up to 12 insts/cycle execute
IA-64 Instruction Format

- Template bits describe grouping of these instructions with others in adjacent bundles
- Each group contains instructions that can execute in parallel
IA-64 Registers

- 128 General Purpose 64-bit Integer Registers
- 128 General Purpose 64/80-bit Floating Point Registers
- 64 1-bit Predicate Registers

- GPRs “rotate” to reduce code size for software pipelined loops
  - Rotation is a simple form of register renaming allowing one instruction to address different physical registers on each iteration
Rotating Register Files

Problems: Scheduled loops require lots of registers, 
Lots of duplicated code in prolog, epilog

Solution: Allocate new set of registers for each loop iteration
Rotating Register Base (RRB) register points to base of current register set. Value added on to logical register specifier to give physical register number. Usually, split into rotating and non-rotating registers.
Rotating Register File
(Previous Loop Example)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RRB Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld P10, ()</td>
<td>RRB=1</td>
</tr>
<tr>
<td>fadd P9, P8,</td>
<td>RRB=2</td>
</tr>
<tr>
<td>ld P9, ()</td>
<td>RRB=3</td>
</tr>
<tr>
<td>fadd P8, P7,</td>
<td>RRB=4</td>
</tr>
<tr>
<td>ld P8, ()</td>
<td>RRB=5</td>
</tr>
<tr>
<td>fadd P7, P6,</td>
<td>RRB=6</td>
</tr>
<tr>
<td>ld P7, ()</td>
<td>RRB=7</td>
</tr>
<tr>
<td>fadd P6, P5,</td>
<td>RRB=8</td>
</tr>
<tr>
<td>ld P6, ()</td>
<td></td>
</tr>
<tr>
<td>fadd P5, P4,</td>
<td></td>
</tr>
<tr>
<td>ld P5, ()</td>
<td></td>
</tr>
<tr>
<td>fadd P4, P3,</td>
<td></td>
</tr>
<tr>
<td>ld P4, ()</td>
<td></td>
</tr>
<tr>
<td>fadd P3, P2,</td>
<td></td>
</tr>
<tr>
<td>ld P3, ()</td>
<td></td>
</tr>
<tr>
<td>fadd P2, P1,</td>
<td></td>
</tr>
<tr>
<td>ld P2, ()</td>
<td></td>
</tr>
<tr>
<td>fadd P1, P0,</td>
<td></td>
</tr>
<tr>
<td>ld P1, ()</td>
<td></td>
</tr>
</tbody>
</table>

Three cycle load latency encoded as difference of 3 in register specifier number (f4 - f1 = 3)

Four cycle fadd latency encoded as difference of 4 in register specifier number (f9 – f5 = 4)
IA-64 Predicated Execution

Problem: Mispredicted branches limit ILP
Solution: Eliminate hard to predict branches with predicated execution
- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false

Four basic blocks

Inst 1
Inst 2
br a==b, b2

b0:

Inst 3
Inst 4
br b3

b1:

Inst 5
Inst 6

b2:

Inst 7
Inst 8

b3:

Predication

Inst 1
Inst 2
p1,p2 <- cmp(a==b)
(p1) Inst 3  ||  (p2) Inst 5
(p1) Inst 4  ||  (p2) Inst 6
Inst 7
Inst 8

One basic block

Mahlke et al, ISCA95: On average >50% branches removed
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion
Solution: Speculative operations that don’t cause exceptions

Inst 1
Inst 2
br a==b, b2

Load r1
Use r1
Inst 3

Can’t move load above branch because might cause spurious exception

Speculative load never causes exception, but sets “poison” bit on destination register

Load.s r1
Inst 1
Inst 2
br a==b, b2

Chk.s r1
Use r1
Inst 3

Check for exception in original home block jumps to fixup code if exception detected

Particularly useful for scheduling long latency loads early
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling
Solution: Hardware to check pointer hazards

Inst 1
Inst 2
Store

Load r1
Use r1
Inst 3

Can’t move load above store because store might be to same address

Data speculative load adds address to address check table

Store invalidates any matching loads in address check table

Load.a r1
Inst 1
Inst 2
Store

Load.c
Use r1
Inst 3

Check if load invalid (or missing), jump to fixup code if so

Requires associative hardware in address check table
Limits of Static Scheduling

- Unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity
- Despite several attempts, VLIW has failed in general-purpose computing arena (so far).
  - More complex VLIW architectures close to in-order superscalar in complexity, no real advantage on large complex apps.
- Successful in embedded DSP market
  - Simpler VLIWs with more constrained environment, friendlier code.
Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)