ECE 4750 Computer Architecture

Topic 13: VLIW Processors

Christopher Batten
School of Electrical and Computer Engineering
Cornell University

http://www.csl.cornell.edu/courses/ece4750
Superscalar Control Logic Scaling

- Each issued instruction must somehow check against $W \times L$ instructions, i.e., growth in hardware $\propto W \times (W \times L)$
- For in-order machines, $L$ is related to pipeline latencies and check is done during issue (scoreboard)
- For out-of-order machines, $L$ also includes time spent in IQ, SB, WQ, and check is done by broadcasting tags to waiting instructions at completion
- As $W$ increases, larger instruction window is needed to find enough parallelism to keep machine busy => greater $L$

=> *Out-of-order control logic grows faster than $W^2$ ($\sim W^3$)*
Out-of-Order Control Complexity:
MIPS R10000

[ SGI/MIPS Technologies Inc., 1995 ]
Sequential ISA Bottleneck

Sequential source code

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

Superscalar compiler

Find independent operations
Schedule operations

Sequential machine code

Superscalar processor

Check instruction dependencies
Schedule execution
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
VLIW Compiler Responsibilities

• Schedules to maximize parallel execution

• Guarantees intra-instruction parallelism

• Schedules to avoid data hazards (no interlocks)
  – Typically separates operations with explicit NOPs
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
Loop Execution

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

Compile

```
loop:
lw f1, 0(r1)
addiu r1, 4
add.s f2, f0, f1
sw f2, 0(r2)
addiu r2, 4
bne r1, r3, loop
```

Schedule

```
<table>
<thead>
<tr>
<th></th>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td>add.s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>add.s</td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sw</td>
<td></td>
</tr>
</tbody>
</table>
```

How many FP ops/cycle?

1 add.s / 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**

```assembly
loop:   lw f1, 0(r1)
        lw f2, 8(r1)
        lw f3, 16(r1)
        lw f4, 24(r1)
        addiu r1, 16
        add.s f5, f0, f1
        add.s f6, f0, f2
        add.s f7, f0, f3
        add.s f8, f0, f4
        sw f5, 0(r2)
        sw f6, 8(r2)
        sw f7, 16(r2)
        sw f8, 24(r2)
        addiu r2, 16
        bne r1, r3, loop
```

**Schedule**

<table>
<thead>
<tr>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1</td>
<td>add.s f5</td>
<td>add.s f6</td>
<td>add.s f7</td>
<td>add.s f8</td>
<td></td>
</tr>
<tr>
<td>add r2</td>
<td>bne</td>
<td>sw f5</td>
<td>sw f6</td>
<td>sw f7</td>
<td>sw f8</td>
</tr>
</tbody>
</table>

**How many FLOPS/cycle?**

\[
4 \text{ add.ss} / 11 \text{ cycles} = 0.36
\]
Loop Unrolling & Software Pipelining

**Unroll 4 ways first**

```
loop:  lw f1, 0(r1)
       lw f2, 8(r1)
       lw f3, 16(r1)
       lw f4, 24(r1)
       addiu r1, 16
       add.s f5, f0, f1
       add.s f6, f0, f2
       add.s f7, f0, f3
       add.s f8, f0, f4
       sw f5, 0(r2)
       sw f6, 8(r2)
       sw f7, 16(r2)
       sw f8, 24(r2)
       addiu r2, 16
       bne r1, r3, loop

iterate
```

```
prolog
```

```
epilog
```

```
How many FLOPS/cycle?
4 add.ss / 4 cycles = 1
```
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
- Only one branch per VLIW instruction
- Difficult to find ILP in individual basic blocks
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
Predication

Problem: Mispredicted branches limit ILP
Solution: Eliminate hard to predict branches with predicated execution

Predication helps with small branch regions and/or branches that are hard to predict by turning control flow into data flow

Most basic form of predication: conditional moves

\[
\text{movz } rd, rs, rt \quad \text{if } ( R[rt] == 0 ) \quad \text{then } R[rd] <- R[rs] \\
\text{movn } rd, rs, rt \quad \text{if } ( R[rt] != 0 ) \quad \text{then } R[rd] <- R[rs] 
\]

\[
\begin{align*}
\text{If } ( a < b ) & \quad \text{slt } r1, r2, r3 & \quad \text{slt } r1, r2, r3 \\
\text{x = a} & \quad \text{beq } r1, r0, L1 & \quad \text{movz } r4, r2, r1 \\
\text{else} & \quad \text{move } r4, r2 & \quad \text{movn } r4, r3, r1 \\
\text{x = b} & \quad \text{j } L2 & \quad \text{} \\
L1: & \quad \text{move } r4, r3 & \quad \text{} \\
L2: & \quad \text{} & \quad \text{What if then else has many instructions? Or is unbalanced?}
\end{align*}
\]
Full Predication

- Almost all instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false

\[
\begin{align*}
\text{b0:} & \quad \text{Inst 1} \quad \text{if} \\
 & \quad \text{Inst 2} \\
 & \quad \text{br a==b, b2} \\
\text{b1:} & \quad \text{Inst 3} \quad \text{else} \\
 & \quad \text{Inst 4} \\
 & \quad \text{br b3} \\
\text{b2:} & \quad \text{Inst 5} \quad \text{then} \\
 & \quad \text{Inst 6} \\
\text{b3:} & \quad \text{Inst 7} \\
 & \quad \text{Inst 8}
\end{align*}
\]

Mahlke et al, ISCA95: On average >50% branches removed
Rotating Register File

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)

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

Four cycle add.s latency encoded as difference of 4 in register specifier number (f9 - f5 = 4)

lw f1, ()  add.s f5, f4, ...  sw f9, ()  bloop

lw P9, ()  add.s P13, P12,  sw P17, ()  bloop
lw P8, ()  add.s P12, P11,  sw P16, ()  bloop
lw P7, ()  add.s P11, P10,  sw P15, ()  bloop
lw P6, ()  add.s P10, P9,  sw P14, ()  bloop
lw P5, ()  add.s P9, P8,  sw P13, ()  bloop
lw P4, ()  add.s P8, P7,  sw P12, ()  bloop
lw P3, ()  add.s P7, P6,  sw P11, ()  bloop
lw P2, ()  add.s P6, P5,  sw P10, ()  bloop
Cydra-5: Memory Latency Register (MLR)

Problem: Loads have variable latency
Solution: Let software choose desired memory latency

- Compiler schedules code for maximum load-use distance
- Software sets MLR to latency that matches code schedule
- Hardware ensures that loads take exactly MLR cycles to return values into processor pipeline
  - Hardware buffers loads that return early
  - Hardware stalls processor if loads return late
Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)

• MIT material derived from course 6.823
• UCB material derived from course CS252 & CS152