Recap: Advanced Superscalars

- Even simple branch prediction can be quite effective
- Path-based predictors can achieve >95% accuracy
- BTB redirects control flow early in pipe, BHT cheaper per entry but must wait for instruction decode
- Branch mispredict recovery requires snapshots of pipeline state to reduce penalty
- Unified physical register file design, avoids reading data from multiple locations (ROB+arch regfile)
- Superscalars can rename multiple dependent instructions in one clock cycle
- Need speculative store buffer to avoid waiting for stores to commit
Recap: Branch Prediction and Speculative Execution

Little's Law

Parallelism = Throughput \times Latency

\bar{N} = \bar{T} \times \bar{L}
**Example Pipelined ILP Machine**

Max Throughput, Six Instructions per Cycle

- How much instruction-level parallelism (ILP) required to keep machine pipelines busy?

\[
T = 6 \quad L = \frac{2 \times 1 + 2 \times 3 + 2 \times 4}{6} = 2 \frac{2}{3} \quad N = 6 \times 2 \frac{2}{3} = 16
\]

**Superscalar Control Logic Scaling**

- Each issued instruction must check against \(W*L\) instructions, i.e., growth in hardware \(\propto W^*(W*L)\)
- For in-order machines, \(L\) is related to pipeline latencies
- For out-of-order machines, \(L\) also includes time spent in instruction buffers (instruction window or ROB)
- As \(W\) increases, larger instruction window is needed to find enough parallelism to keep machine busy \(\Rightarrow\) greater \(L\)

\(\Rightarrow\) **Out-of-order control logic grows faster than** \(W^2 (~W^3)\)
Out-of-Order Control Complexity: MIPS R10000

Sequential ISA Bottleneck

Sequential source code

Superscalar compiler

Find independent operations

Schedule operations

Sequential machine code

Superscalar processor

Check instruction dependencies

Schedule execution

9/18/2007
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 x-operation RAW check
  - No data use before data ready => no data interlocks

Two Integer Units, Single Cycle Latency
Two Load/Store Units, Three Cycle Latency
Two Floating-Point Units, Four Cycle Latency

VLIW Compiler Responsibilities

The compiler:

- 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

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

Compile

loop:
  ld f1, 0(r1)
  add r1, 8
  fadd f2, f0, f1
  sd f2, 0(r2)
  add r2, 8
  bne r1, r3, loop

Schedule

How many FP ops/cycle?

1 fadd / 8 cycles = 0.125
Loop Unrolling

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

Unroll inner loop to perform 4 iterations at once

```c
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:**
```
ld f1, 0(r1)
ld f2, 8(r1)
ld f3, 16(r1)
ld f4, 24(r1)
add r1, 32
fadd f5, f0, f1
fadd f6, f0, f2
fadd f7, f0, f3
fadd f8, f0, f4
sd f5, 0(r2)
sd f6, 8(r2)
sd f7, 16(r2)
sd f8, 24(r2)
add r2, 32
bne r1, r3, loop
```

**Schedule**

```

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36
```
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

Trace Scheduling \[ \text{Fisher,Ellis} \]

- 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 l-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
### Rotating Register Files

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

```
<table>
<thead>
<tr>
<th></th>
<th>Prolog</th>
<th>Loop</th>
<th>Epilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r1, ()</td>
<td>ld r1, ()</td>
<td>add r2, r1, #1</td>
<td>st r2, ()</td>
</tr>
<tr>
<td>add r2, r1, #1</td>
<td></td>
<td></td>
<td>add r2, r1, #1</td>
</tr>
<tr>
<td>st r2, ()</td>
<td></td>
<td></td>
<td>st r2, ()</td>
</tr>
<tr>
<td>st r2, ()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Solution: Allocate new set of registers for each loop iteration.

### Rotating Register File

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.

```
<table>
<thead>
<tr>
<th></th>
<th>Prolog</th>
<th>Loop</th>
<th>Epilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r1, ()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld r1, ()</td>
<td>add r3, r2, #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r3, r2, #1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st r4, ()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st r4, ()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Loop closing branch decrements RRB.
Rotating Register File
(Previous Loop Example)

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

```
ld f1, ()  fadd f5, f4, ...  sd f9, ()  bloop
```

Four cycle fadd latency encoded as difference of 4 in register specifier number (f9 – f5 = 4)

```
ld P9, ()  fadd P13, P12,  sd P17, ()  bloop
ld P8, ()  fadd P12, P11,  sd P16, ()  bloop
ld P7, ()  fadd P11, P10,  sd P15, ()  bloop
ld P6, ()  fadd P10, P9,   sd P14, ()  bloop
ld P5, ()  fadd P9, P8,    sd P13, ()  bloop
ld P4, ()  fadd P8, P7,    sd P12, ()  bloop
ld P3, ()  fadd P7, P6,    sd P11, ()  bloop
ld P2, ()  fadd P6, P5,    sd P10, ()  bloop
```

RRB=8
RRB=7
RRB=6
RRB=5
RRB=4
RRB=3
RRB=2
RRB=1

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
CS252 Administrivia

- Project proposal due one week from today
- Title, team members’ names, one page PDF writeup
- Send matchmaking email to class if you don’t have partner
- Krste office hours 1-3pm, Monday 645 Soda

Intel EPIC IA-64

- EPIC is the style of architecture (cf. CISC, RISC)
  - Explicitly Parallel Instruction Computing
- IA-64 is Intel’s chosen ISA (cf. x86, MIPS)
  - IA-64 = Intel Architecture 64-bit
  - An object-code compatible VLIW
- Itanium (aka Merced) is first implementation (cf. 8086)
  - First customer shipment expected 1997 (actually 2001)
  - McKinley, second implementation shipped in 2002
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.

128-bit instruction bundle

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.
**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

```
b0:  if
     Inst 1
     Inst 2
     br a==b, b2
b1:  else
     Inst 3
     Inst 4
     br b3
b2:  then
     Inst 5
     Inst 6
b3:  Inst 7
     Inst 8
```

*Mahlke et al, ISCA95: On average >50% branches removed*

**Predicate Software Pipeline Stages**

**Single VLIW Instruction**

```
(p1) ld r1   (p2) add r3   (p3) st r4   (p1) bloop
```

**Dynamic Execution**

```
(p1) ld r1
(p1) ld r1 (p2) add r3 (p1) bloop
(p1) ld r1 (p2) add r3 (p1) bloop (p3) st r4
(p1) ld r1 (p2) add r3 (p1) bloop (p3) st r4 (p3) st r4
(p2) add r3 (p1) bloop (p3) st r4 (p3) st r4 (p1) bloop
```

Software pipeline stages turned on by rotating predicate registers ➔ Much denser encoding of loops
Fully Bypassed Datapath

Where does predication fit in?

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

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

Chk.s r1
Use r1
Inst 3

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

Requires associative hardware in address check table

Clustered VLIW

- Divide machine into clusters of local register files and local functional units
- Lower bandwidth/higher latency interconnect between clusters
- Software responsible for mapping computations to minimize communication overhead
- Common in commercial embedded VLIW processors, e.g., TI C6x DSPs, HP Lx processor
- (Same idea used in some superscalar processors, e.g., Alpha 21264)
Limits of Static Scheduling

- Unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity

Question:

How applicable are the VLIW-inspired techniques to traditional RISC/CISC processor architectures?

Paper Discussion: CISC vs RISC

Recommended optional further reading:

- D. Bhandarkar and D. W. Clark. “Performance from architecture: Comparing a RISC and a CISC with similar hardware organization”, In Intl. Conf. on Architectural Support for Prog. Lang. and Operating Sys., ASPLOS-IV, Santa Clara, CA, Apr. 1991, pages 310--319 - conclusion is RISC is 2.7x better than CISC!
Paper Discussion: Wall ILP Study

What conclusions do you draw?
What are biggest limitations of study?