Microcoding, VLIW, and Vector Computers

Ryan Lee
Adapted from Suvinay Subramanian
6.823 Spring 2019
Microcoding

» Abstraction layer between hardware and architecture of computer
  - i.e. separates ISA from actual hardware implementation details.

» Layer of hardware-level instructions that implement higher-level (i.e. ISA) instructions
Benefits of Microcoding

» Originally developed as a simpler method for developing control logic
  - Direct combinational logic for powerful instruction set can be complex, prone to bugs/difficult to debug
  - Multi-step addressing modes, varied length instructions etc.

» Enables complex ISAs
  - Reduced instruction fetch bandwidth; smaller code footprint

» Same ISA, different implementations
  - Flexibility in how data paths, micro-arch. blocks designed
  - IBM 360: Model 30, Model 40 etc.

» Allows for “patching” in the field

» New instructions supported without modifying datapath

Are there any downsides though?
Microcode Implementation

- Opcode
- ext
- op-group
- absolute
- μPC
- μPC+1
- input encoding reduces ROM height
- μPC (state)
- +1
- μPCSrc
- jump logic
- zero
- busy
- μJumpType = next | spin
- | fetch | dispatch
- | feqz | fnez
- Control ROM
- data
- address
- Control Signals (17)
# Microcode Fragments

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch₀</td>
<td>MA ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch₁</td>
<td>IR ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch₂</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch₃</td>
<td>PC ← A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU₁</td>
<td>B ← Reg[rt]</td>
<td>next</td>
</tr>
<tr>
<td>ALU₂</td>
<td>Reg[rd]←func(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>ALUi₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALUi₁</td>
<td>B ← sExt₁₆(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>ALUi₂</td>
<td>Reg[rd]← Op(A,B)</td>
<td>fetch</td>
</tr>
</tbody>
</table>
VLIW

» Motivation: OoO processors introduce complex, inefficient hardware for uncovering ILP

» Premise: Static instruction scheduling + super-scalar execution to extract ILP.

» Tradeoff: Complex hardware vs Complex compiler
  - OoO processors do dynamic scheduling
    Figure out independent instructions on-the-fly
  - VLIW machines: Compilers figure out independent instructions and schedules them suitably
VLIW Motivation

From Cornell University ECE 4750 Handout #15, Courtesy Chris Batten and course staff
VLIW Hardware

- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
VLIW Software

» Key Questions:
  - How do we find independent instructions to fetch/execute?
  - How to enable more compiler optimizations?

» Key Ideas:
  - Get rid of control flow
    • Predicated execution, loop unrolling
  - Optimize frequently executed code-paths
    • Trace scheduling
  - Others: Software pipelining
Loop Unrolling

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

Compile

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

Schedule

loop:

Int1  Int2  M1  M2  FP+
-----------
add r1       ld
    fadd
add r2       bne
    sd

How many FP ops/cycle?

1 fadd / 8 cycles = 0.125
Loop Unrolling

» Unroll loop to perform M iterations at once
  - Get more independent instructions
  - Need to be careful about case where M is not a multiple of number of loop iterations

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

```c
for (i=0; i<N; i+=4)
{
}
```
Loop Unrolling

**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>ld f1</td>
<td></td>
<td>fadd f5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f2</td>
<td></td>
<td>fadd f6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f3</td>
<td></td>
<td>fadd f7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f4</td>
<td></td>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1, 32</td>
<td>sd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2</td>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 fadd operations / 11 cycles = 0.36
Loop Unrolling

1. Combine M iterations of loop
2. Pipeline schedule to reduce RAW stalls
   - In the example above, notice that we move (re-order) loads to the top
3. Rename registers
   - f1, f2, f3, f4
Software Pipelining

```
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)
       add r2, 32
       sd f8, -8(r2)
       bne r1, r3, loop
```

```
prolog

iterate

loop:

epilog

```

```

4 fadds / 4 cycles = 1
Loop Unrolling Limitations

» Code growth
» Does not handle inter-iteration dependences well
Predicated Execution

» Limited ILP within a basic-block; branches limit available ILP

» Idea: Eliminate hard-to-predict branches by converting control dependence to data dependence
  - Each instruction (within the branch basic block) has a predicate bit set
  - Only instructions with true predicates are executed and committed. Others are treated as nops.
Predicated Execution.

Four basic blocks

Predication

One basic block

Inst 1
Inst 2
br a==b, b2

Inst 3
Inst 4
br b3

Inst 5
Inst 6

Inst 7
Inst 8

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
Speculative Execution

» Move instructions above branches to explore more ILP options
Speculative Execution

```c
int32_t calculateSomething(int32_t *a, int32_t *b) {
    int32_t result;
    if (m_p > m_q + 1) {
        result = a[*b];
    } else {
        result = defaultValue;
    }
    return result;
}
```

Speculative execute load of $a[*b]$ before branch condition is resolved. Say: $m_p$, $m_q$ are bound checks.

1. What if $m_p < m_q + 1$? Say, $b = nullptr$. Exception!
2. But exceptions can arise in other ways. What if $a[*b]$ is valid, but there is a page-fault? Exception! Trap to OS routine.
Trace Scheduling

» Idea: For non-loop situations:
  - Find common path in program trace
  - Re-align basic blocks to form straight-line trace
    • Trace: Fused basic-block sequence
  - Schedule trace
  - Create fixup code in case trace != actual path
    • Can be nasty
Trace Scheduling

\[ A = Y[i] \]
\[ \text{If} \ (A == 0) \]
\[ \quad A = W[i] \]
\[ \text{Else:} \]
\[ \quad Y[i] = 0 \]
\[ Z[i] = A \times X[i] \]

#0: ldf f2,Y(r1)
#1: bfnez f2,instr#4
#2: ldf f2,W(r1)
#3: jump instr#5
#4: stf f0,Y(r1)
#5: ldf f4,X(r1)
#6: mulf f6,f4,f2
#7: stf f6,Z(r1)
Trace Scheduling

A
#0: ldf f2,Y(r1)
#1: bfnez f2,instr#4

B
#2: ldf f2,W(r1)
#3: jump instr#5

NT: 10%

C
#4: stf f0,Y(r1)

T: 90%

D
#5: ldf f4,X(r1)
#6: mulf f6,f4,f2
#7: stf f6,Z(r1)
Trace Scheduling

#0: ldf f2,Y(r1)
#1: bfeqz f2,#2
#4: stf f0,Y(r1)
#5: ldf f4,X(r1)
#6: mulf f6,f4,f2
#7: stf f6,Z(r1)

Recovery / repair code

#2 : ldf f2,W(r1)
#5’: ldf f4,X(r1)
#6’: mulf f6,f4,f2
#7’: stf f6,Z(r1)

A, C, D superbloc (trace)

- Trace scheduling can be combined with other techniques:
  - What if we moved #5, #6 before #4? Speculative load issue
  - What if branch was biased the other way?
  - What if branch was evenly biased – 50%, 50%? Predication
VLIW Summary

» Loop unrolling
  - Reduces branch frequency
  - Tighter packing of instructions
  - Dependences b/w iterations; handling “extra” iterations

» Predicated execution, speculative execution
  - Control-flow
  - Control-flow, Load-store speculation

» Trace scheduling
  - Recovery code
  - Combined with other techniques above; moving code upward/downward may provide benefits
Vector Computers

» Idea: Operate on vectors instead of scalars
  - ISA is more expressive, therefore captures more information

» Advantages:
  - No dependences within a vector
  - Reduced instruction fetch bandwidth
  - Amortized cost of instruction fetch and decode
  - (Sometimes) regular memory access pattern
  - No need to explicitly code loops

» Pitfalls:
  - Only works if code sequence (or parallelism) is regular
Vector Computers

Terminology:
» Vector length register (VLR)
» Conditional execution using vector mask (VM)
» Vector lanes
» Vector chaining
Vector Computers

LV v1
MULV v3,v1,v2
ADDV v5, v3, v4

Load Unit
Memory

Chain

V1
V2
V3
V4
V5

Load
Mul
Add

Load
Mul
Add

Mult.
Add