Quiz 4 Review

Ryan Lee
Adapted from Suvinay Subramanian
6.823 Spring 2019
Topics Snapshot

» Microcoding

» VLIW
  - Loop unrolling, software pipelining, predicated execution, speculative execution, trace scheduling

» Vector Computers
  - Vector lanes, vector length register, masking, chaining

» GPUs
  - Warps, branch divergence (masking)

» Transactional Memory
  - Eager, lazy versioning
  - Optimistic, pessimistic conflict detection

» Virtualization
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
Microcode Implementation

μJumpType = next | spin | fetch | dispatch | feqz | fnez
## 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>
Topics Snapshot

» Microcoding

» VLIW
  - Loop unrolling, software pipelining, predicated execution, speculative execution, trace scheduling

» Vector Computers
  - Vector lanes, vector length register, masking, chaining

» GPUs
  - Warps, branch divergence (masking)

» Transactional Memory
  - Eager, lazy versioning
  - Optimistic, pessimistic conflict detection

» Virtualization
VLIW

» Premise: Static instruction scheduling + superscalar execution to extract ILP.

» Tradeoff: Complex hardware vs Complex compiler “Conservation of complexity”
  - VLIW machines: Compilers figure out independent instructions and schedules them suitably
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, speculative execution
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)
{
}
```
You should be familiar with filling in such a table
**Software Pipelining**

Execute different iterations in parallel.
Predicated, Speculative Execution

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

» Predication: 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.

» Speculation: Move instructions above branches to explore more ILP options
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
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%
T: 90%

C

#4: stf f0,Y(r1)

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

D
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)

Push branch below?
Topics Snapshot

» Microcoding

» VLIW
  - Loop unrolling, software pipelining, predicated execution, speculative execution, trace scheduling

» Vector Computers
  - Vector lanes, vector length register, masking, chaining

» GPUs
  - Warps, branch divergence (masking)

» Transactional Memory
  - Eager, lazy versioning
  - Optimistic, pessimistic conflict detection

» Virtualization
Vector Computers

» Single-instruction multiple data (SIMD)
  - Single instruction to operate on an entire vector (instead of scalars)

» Vector length register (VLR)
  Vector masking (conditional execution)
  Vector chaining
  Vector lanes

If we ask code, we will provide syntax, meaning for vector instructions
GPUs

» Single-instruction multiple thread (SIMT)
  - Multiple instruction streams of scalar instructions

» Warps: A set of threads grouped together, and executing the same instruction (modulo divergence)

» Branch divergence: Handled through masking
Topics Snapshot

» Microcoding

» VLIW
  - Loop unrolling, software pipelining, predicated execution, speculative execution, trace scheduling

» Vector Computers
  - Vector lanes, vector length register, masking, chaining

» GPUs
  - Warps, branch divergence (masking)

» Transactional Memory
  - Eager, lazy versioning
  - Optimistic, pessimistic conflict detection

» Virtualization
Transactional Memory

» Idea: No locks, only shared data
  Idea: Optimistic (speculative) concurrency
  - Execute critical section speculatively
  - Abort on conflicts

» Key properties:
  - Atomicity (all or nothing)
  - Isolation (no other code can observe updates before commit)
  - Serializability
Transactional Memory Taxonomy

» Data Versioning
  - Eager
  - Lazy

» Conflict Detection
  - Pessimistic
  - Optimistic
Data Management Policy

1. Eager versioning (undo-log based)
   - Update memory location directly
   - Maintain undo info in a log
   - Fast commits
   - Slow aborts

2. Lazy versioning (write-buffer based)
   - Buffer data until commit in a write buffer
   - Update actual memory locations at commit
   - Fast aborts
   - Slow commits

How to manage the “tentative work” that a transaction does
Conflict Detection Policy

1. Pessimistic detection
   Check for conflicts during loads or stores

2. Optimistic detection
   Detect conflicts when a transaction attempts to commit
Topics Snapshot

» Microcoding

» VLIW
  - Loop unrolling, software pipelining, predicated execution, speculative execution, trace scheduling

» Vector Computers
  - Vector lanes, vector length register, masking, chaining

» GPUs
  - Warps, branch divergence (masking)

» Transactional Memory
  - Eager, lazy versioning
  - Optimistic, pessimistic conflict detection

» Virtualization
Virtualization

» Goals of an Operating System (OS)
   - Protection and privacy
   - Abstraction (ABI)
   - Resource management

» Same logic for supporting multiple OSs with a VMM.

![Diagram showing process flow and layers in virtualization](image-url)