Announcements

- Project Poster Session
  - December 10
  - NSH Atrium
    - 2:30-6:30pm

- Project Report Due
  - December 12
  - The report should be like a good conference paper

- Focus on Projects
  - All group members should contribute
  - Use the milestone feedback from the TAs
Final Project Report and Logistics

- Follow the guidelines in project handout
  - We will provide the Latex format

- Good papers should be similar to the best conference papers you have been reading throughout the semester

- Submit all code, documentation, supporting documents and data
  - Provide instructions as to how to compile and use your code
  - This will determine part of your grade

- This is the single most important part of the project
Best Projects

- Best projects will be encouraged for a top conference submission
  - Talk with me if you are interested in this

Examples from past:

- Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," HPCA 2010 Best Paper Session
- George Nychis, Chris Fallin, Thomas Moscibroda, and Onur Mutlu, "Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need?,” HotNets 2010.
- Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior,” MICRO 2010. (IEEE Micro Top Picks 2010)
Today

- Alternative approaches to concurrency
  - SI SD/SI MD/MI SD/MI MD classification
  - Decoupled Access/Execute
  - VLIW
  - Vector Processors and Array Processors
  - Data Flow
Alternative Approaches to Concurrency
Readings

Required:

Recommended:
Decoupled Access/Execute

- **Motivation:** Tomasulo’s algorithm too complex to implement
  - 1980s before HPS, Pentium Pro

- **Idea:** Decouple operand access and execution via two separate instruction streams that communicate via ISA-visible queues.

Loop Unrolling

i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    i = i + 1
}

---

- **Idea:** Replicate loop body multiple times within an iteration
- Reduces loop maintenance overhead
  - Induction variable increment or loop condition test
- Enlarges basic block (and analysis scope)
  - Enables code optimization and scheduling opportunities

-- What if iteration count not a multiple of unroll factor? (need extra code to detect this)
-- Increases code size
VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
  - Packed instructions can be logically unrelated (contrast with SIMD)

- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction

- Traditional Characteristics
  - Multiple functional units
  - Each instruction in a bundle executed in lock step
  - Instructions in a bundle statically aligned to be directly fed into the functional units

- ELI: Enormously longword instructions (512 bits)
SIMD Array Processing vs. VLIW

- Array processor

Program Counter \[ \text{add VR, VR, 1} \]

VLEN = 4

Instruction Execution

PE

add VR[0], VR[0], 1

add VR[1], VR[1], 1

add VR[2], VR[2], 1

add VR[3], VR[3], 1
VLIW Philosophy

- Philosophy similar to RISC (simple instructions)
  - Except multiple instructions in parallel

- RISC (John Cocke, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding → very simple

- VLIW (Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple and streamlined as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design
Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors)
  - Most successful commercially

- Intel IA-64
  - Not fully VLIW, but based on VLIW principles
  - EPIC (Explicitly Parallel Instruction Computing)
  - Instruction bundles can have dependent instructions
  - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones
VLIW Tradeoffs

- **Advantages**
  - + No need for dynamic scheduling hardware → simple hardware
  - + No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming
  - + No need for instruction alignment/distribution after fetch to different functional units → simple hardware

- **Disadvantages**
  -- Compiler needs to find N independent operations
    -- If it cannot, inserts NOPs in a VLIW instruction
    -- Parallelism loss AND code size increase
  -- Recompilation required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
  -- Lockstep execution causes independent operations to stall
    -- No instruction can progress until the longest-latency instruction completes
VLIW NOPs

- Cause code bloat + reduce performance
  - Early VLIW machines suffered from this (Multiflow, Cydrome)

- Modern EPIC machines use compaction encoding
- Idea: Encode the existence or lack of NOPs rather than explicitly inserting NOPs

- VLIW Instruction: Variable-length bundles of instructions
- Instruction: Fixed length
- Instruction format
  - Header bit (cycle starting with this operation)
  - Operation type (dispersement)
  - Pause specifier (cycles of NOPs inserted after current cycle)
VLIW NOP Encoding


```
+ No NOPs in the code or I-cache
-- Variable length VLIW instructions, more work decoding
-- Does not eliminate the performance degradation of NOPs
Precursor to IA-64 instruction encodings
```

Latency of A: 2 cycles
Static Instruction Scheduling

- What does the compiler need to know?

- For VLIW scheduling and instruction formation
  - VLIW width
  - Functional unit types and organization
  - Functional unit latencies

- For scheduling in superscalar, in-order processors
  - Superscalar width
  - Functional unit latencies
VLIW: Finding Independent Operations

- Within a basic block, there is limited instruction-level parallelism
- To find multiple instructions to be executed in parallel, the compiler needs to consider multiple basic blocks

- Problem: Moving instructions above a branch is unsafe because instruction is not guaranteed to be executed

- Idea: Enlarge basic blocks at compile time by finding the frequently-executed paths
  - Trace scheduling
  - Superblock scheduling (we’ve already seen the basic idea)
  - Hyperblock scheduling
Safety and Legality in Code Motion

- Two characteristics of speculative code motion:
  - Safety: whether or not spurious exceptions may occur
  - Legality: whether or not result will be correct always

- Four possible types of code motion:

(a) safe and legal

(b) illegal

(c) unsafe

(d) unsafe and illegal
Code Movement Constraints

- **Downward**
  - When moving an operation from a BB to one of its dest BB’s,
    - all the other dest basic blocks should still be able to use the result of the operation
    - the other source BB’s of the dest BB should not be disturbed

- **Upward**
  - When moving an operation from a BB to its source BB’s
    - register values required by the other dest BB’s must not be destroyed
    - the movement must not cause new exceptions
Trace Scheduling

- Trace: A frequently executed path in the control-flow graph (has multiple side entrances and multiple side exits)

- Idea: Find independent operations within a trace to pack into VLIW instructions.
  - Traces determined via profiling
  - Compiler adds fix-up code for correctness (if a side entrance or side exit of a trace is exercised at runtime)
There may be conditional branches from the middle of the trace (side exits) and transitions from other traces into the middle of the trace (side entrances).

These control-flow transitions are ignored during trace scheduling.

After scheduling, bookeeping code is inserted to ensure the correct execution of off-trace code.

Trace Scheduling Idea

TRACE SCHEDULING LOOP-FREE CODE
Trace Scheduling (III)

What bookkeeping is required when Instr 1 is moved below the side entrance in the trace?
Trace Scheduling (IV)
Trace Scheduling (V)

<table>
<thead>
<tr>
<th></th>
<th>Instr 1</th>
<th>Instr 2</th>
<th>Instr 3</th>
<th>Instr 4</th>
<th>Instr 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What bookkeeping is required when Instr 5 moves above the side entrance in the trace?
Trace Scheduling (VI)
**Trace Scheduling Fixup Code Issues**

- Sometimes need to copy instructions more than once to ensure correctness on all paths (see C below)

![Diagram](image)

**Original trace**
- A → B → C → D → E
- X → Y

**Scheduled trace**
- D → B → A → E → C
- C → C’ → Y

**Correctness**
- B → X
- C
- D → Y

![Correctness Diagram](image)
Trace Scheduling Overview

- **Trace Selection**
  - select seed block (the highest frequency basic block)
  - extend trace (along the highest frequency edges)
    - forward (successor of the last block of the trace)
    - backward (predecessor of the first block of the trace)
  - don’t cross loop back edge
  - bound max_trace_length heuristically

- **Trace Scheduling**
  - build data precedence graph for a whole trace
  - perform list scheduling and allocate registers
  - add compensation code to maintain semantic correctness

- **Speculative Code Motion (upward)**
  - move an instruction above a branch if safe
Data Precedence Graph
List Scheduling

- Assign priority to each instruction
- Initialize ready list that holds all ready instructions
  - Ready = data ready and can be scheduled
- Choose one ready instruction \( I \) from ready list with the highest priority
  - Possibly using tie-breaking heuristics
- Insert \( I \) into schedule
  - Making sure resource constraints are satisfied
- Add those instructions whose precedence constraints are now satisfied into the ready list
Instruction Prioritization Heuristics

- Number of descendants in precedence graph
- Maximum latency from root node of precedence graph
- Length of operation latency
- Ranking of paths based on importance
- Combination of above
VLIW List Scheduling

- Assign Priorities
- Compute Data Ready List - all operations whose predecessors have been scheduled.
- Select from DRL in priority order while checking resource constraints
- Add newly ready operations to DRL and repeat for next instruction
Trace Scheduling Example (I)

```
beq  r1, $0
fdiv  f1, f2, f3
fadd  f4, f1, f5
add  r2, r2, 4
ld  r2, 0(r3)
beq  r2, $0
add  r8, r8, 4
fsub  f2, f3, f7
st.d  f2, 0(r8)
add  r3, r3, 4
```

9 stalls

```
add  r2, r2, 4
beq  r2, $0
fsub  f2, f2, f6
st.d  f2, 0(r8)
add  r3, r3, 4
add  r8, r8, 4
```

1 stall

```
ld  r2, 0(r3)
```

1 stall

```
ld  r2, 4(r3)
```

1 stall

```
add  r2, r2, 4
beq  r2, $0
add  r2, 0(r3)
add  r2, r2, 4
beq  r2, $0
```
Trace Scheduling Example (II)

```
fdiv  f1,  f2,  f3
beq  r1,  $0

ld  r2, 0(r3)
fsup  f2,  f2,  f6
add  r2,  r2,  4
beq  r2,  $0

fsub  f2,  f2,  f6
st.d  f2, 0(r8)
add  r3,  r3,  4
add  r8,  r8,  4
fadd  f4,  f1,  f5
```
Trace Scheduling Example (III)

fdiv f1, f2, f3
beq r1, $0
ld r2, 0(r3)
fsub f2, f2, f6
add r2, r2, 4
beq r2, $0
st.d f2, 0(r8)
add r3, r3, 4
add r8, r8, 4
fadd f4, f1, f5

Split
comp. code

fadd f4, f1, f5

Join comp. code

B3

B6
Trace Scheduling Example (IV)

```
fdiv f1, f2, f3
beq r1, $0

ld r2, 0(r3)
fsub f2, f2, f6
add r2, r2, 4
beq r2, $0

st.d f2, 0(r8)

add r3, r3, 4
add r8, r8, 4
fadd f4, f1, f5

fadd f4, f1, f5
```

**Split**

```
add r2, r2, 4
beq r2, $0
fsub f2, f2, f6
st.d f2, 0(r8)
add r3, r3, 4
add r8, r8, 4

B3
```

**Copied split instructions**

```
add r3, r3, 4
add r8, r8, 4

B6
```

**Join comp. code**

```
add r3, r3, 4
add r8, r8, 4
```
Trace Scheduling Example (V)

```
fdiv f1, f2, f3
beq r1, $0

ld  r2, 0(r3)
subb f2, f2, f6
add  r2, r2, 4
beq  r2, $0

fsub  f2,  f2,  f6
st.d  f2, 0(r8)

fadd  f4, f1, f5
add  r3, r3, 4
add  r8, r8, 4
add  r3, r3, 4
add  r8, r8, 4
```

Diagram: 
- B3
  - ld  r2, 0(r3)
  - fadd  f4, f1, f5
  - add  r2, r2, 4
  - beq  r2, $0
- B6
  - fadd  f4, f1, f5
  - fsub  f2, f3, f7
  - add  r3, r3, 4
  - add  r8, r8, 4
  - add  r3, r3, 4
  - add  r8, r8, 4
- fsub  f2, f2, f6
  - std  f2, 0(r8)
  - add  r3, r3, 4
  - add  r8, r8, 4
Trace Scheduling Tradeoffs

- Advantages
  + Enables the finding of more independent instructions → fewer NOPs in a VLIW instruction

- Disadvantages
  -- Profile dependent
    -- What if dynamic path deviates from trace → lots of NOPs in the VLIW instructions
  -- Code bloat and additional fix-up code executed
    -- Due to side entrances and side exits
      -- Infrequent paths interfere with the frequent path
  -- Effectiveness depends on the bias of branches
    -- Unbiased branches → smaller traces → less opportunity for finding independent instructions
Superblock Scheduling

- Trace: multiple entry, multiple exit block
- Superblock: single-entry, multiple exit
  - A trace with side entrances are eliminated
  - Infrequent paths do not interfere with the frequent path

+ More optimization/scheduling opportunity than traces
+ Eliminates “difficult” bookkeeping due to side entrances
Can You Do This with a Trace?

Original Code

```
opA: mul r1, r2, 3
```

```
opB: add r2, r2, 1
```

```
opC: mul r3, r2, 3
```

99

Code After Superblock Formation

```
opA: mul r1, r2, 3
```

```
opB: add r2, r2, 1
```

```
opC: mov r3, r1
```

```
opC': mul r3, r2, 3
```

99

Code After Common Subexpression Elimination

```
opA: mul r1, r2, 3
```

```
opB: add r2, r2, 1
```

```
opC': mul r3, r2, 3
```

1
Superblock Scheduling Shortcomings

-- Still profile-dependent

-- No single frequently executed path if there is an unbiased branch
  -- Reduces the size of superblocks

-- Code bloat and additional fix-up code executed
  -- Due to side exits
Hyperblock Scheduling

- **Idea:** Use predication support to eliminate unbiased branches and increase the size of superblocks

- **Hyperblock:** A single-entry, multiple-exit block with internal control flow eliminated using predication (if-conversion)

**Advantages**

- Reduces the effect of unbiased branches on scheduling block size

**Disadvantages**

- Requires predicated execution support
- All disadvantages of predicated execution
Hyperblock Formation (I)

- Hyperblock formation
  1. Block selection
  2. Tail duplication
  3. If-conversion

- Block selection
  - Select subset of BBs for inclusion in HB
  - Difficult problem
  - Weighted cost/benefit function
    - Height overhead
    - Resource overhead
    - Dependency overhead
    - Branch elimination benefit
    - Weighted by frequency

Hyperblock Formation (II)

Tail duplication same as with Superblock formation
Hyperblock Formation (III)

If-convert (predicate) intra-hyperblock branches

\[ p1, p2 = \text{CMPP} \]

- BB1
- BB2
- BB3
- BB4
- BB5
- BB6
- BB6’
Can We Do Better?

- Hyperblock still
  - Profile dependent
  - Requires fix-up code
  - And, requires predication support

- Single-entry, single-exit enlarged blocks
  - Block-structured ISA
    - Optimizes multiple paths (can use predication to enlarge blocks)
    - No need for fix-up code (duplication instead of fixup)
VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
- VLIW architectures have not been commercially successful in the general-purpose computing market. Why?
  -- Too many NOPs (not enough parallelism discovered)
  -- Static schedule intimately tied to microarchitecture
    -- Code optimized for one generation performs poorly for next
  -- No tolerance for variable or long-latency operations (lock step)

- Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
  - Enable code optimizations
EPIC – Intel IA-64 Architecture

- Gets rid of lock-step execution of instructions within a VLIW instruction
- Idea: More ISA support for static scheduling and parallelization
  - Specify dependencies within and between VLIW instructions (explicitly parallel)

+ No lock-step execution
+ Static reordering of stores and loads + dynamic checking
-- Hardware needs to perform dependency checking (albeit aided by software)
-- Other disadvantages of VLIW still exist

IA-64 Instructions

- IA-64 “Bundle” (~EPIC Instruction)
  - Total of 128 bits
  - Contains three IA-64 instructions
  - Template bits in each bundle specify dependencies within a bundle

- IA-64 Instruction
  - Fixed-length 41 bits long
  - Contains three 7-bit register specifiers
  - Contains a 6-bit field for specifying one of the 64 one-bit predicate registers
IA-64 Instruction Bundles and Groups

- Groups of instructions can be executed safely in parallel
  - Marked by template bits

- Bundles are for packaging
  - Groups can span multiple bundles
    - Alleviates recompilation need somewhat
Template Bits

- Specify two things
  - Stop information: Boundary of independent instructions
  - Functional unit information: Where should each instruction be routed

<table>
<thead>
<tr>
<th>Template</th>
<th>Slot 0</th>
<th>Slot 1</th>
<th>Slot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>M-unit</td>
<td>I-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>01</td>
<td>M-unit</td>
<td>I-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>02</td>
<td>M-unit</td>
<td>I-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>03</td>
<td>M-unit</td>
<td>I-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>04</td>
<td>M-unit</td>
<td>L-unit</td>
<td>X-unit</td>
</tr>
<tr>
<td>05</td>
<td>M-unit</td>
<td>L-unit</td>
<td>X-unit</td>
</tr>
<tr>
<td>06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>M-unit</td>
<td>M-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>09</td>
<td>M-unit</td>
<td>M-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>0A</td>
<td>M-unit</td>
<td>M-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>0B</td>
<td>M-unit</td>
<td>M-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>0C</td>
<td>M-unit</td>
<td>F-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>0D</td>
<td>M-unit</td>
<td>F-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>0E</td>
<td>M-unit</td>
<td>M-unit</td>
<td>F-unit</td>
</tr>
<tr>
<td>0F</td>
<td>M-unit</td>
<td>M-unit</td>
<td>F-unit</td>
</tr>
<tr>
<td>10</td>
<td>M-unit</td>
<td>I-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>11</td>
<td>M-unit</td>
<td>I-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>12</td>
<td>M-unit</td>
<td>B-unit</td>
<td>B-unit</td>
</tr>
</tbody>
</table>
Non-Faulting Loads and Exception Propagation

- \textit{ld.s} fetches \textit{speculatively} from memory
  - i.e. any exception due to \textit{ld.s} is suppressed

- If \textit{ld.s r} did not cause an exception then \textit{chk.s r} is an NOP, else a branch is taken (to some compensation code)
Speculatively load data can be consumed prior to check.

“speculation” status is propagated with speculated data.

Any instruction that uses a speculative result also becomes speculative itself (i.e. suppressed exceptions).

*chk.s* checks the entire dataflow sequence for exceptions.
Aggressive ST-LD Reordering in IA-64

- `ld.a` starts the monitoring of any store to the same address as the advanced load.
- If no aliasing has occurred since `ld.a, ld.c` is a NOP.
- If aliasing has occurred, `ld.c` re-loads from memory.

```
inst 1
inst 2
....
st[?]
....
ld r1=[x]
use=r1
```

```
ld.a r1=[x]
inst 1
inst 2
....
st [?]
....
ld.c r1=[x]
use=r1
```
Aggressive ST-LD Reordering in IA-64

potential aliasing

\[
\text{inst 1} \\
\text{inst 2} \\
\ldots \\
\text{ld } r1=[x] \\
\text{use}=r1 \\
\text{st} [?] \\
\ldots
\]

\[
\text{ld } r1=[a] \\
\text{use}=r1
\]

\[
\text{ld.a } r1=[x] \\
\text{inst 1} \\
\text{inst 2} \\
\text{use}=r1 \\
\ldots \\
\text{st } [?] \\
\ldots \\
\text{chk.a } X \\
\ldots
\]
Midterm II Grade Distribution

Midterm II scores

0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180

58