Lecture 9

VLIW & Statically Scheduled ILP

Department of Electrical Engineering
Stanford University

http://eeclasse.stanford.edu/ee382a
Announcements

- HW2 due on Wed 10/21
  - You should be able to solve all problems now

- Readings for this lecture
Review: Memory Data Flow

- Issue #1: stores cannot update memory until they commit from ROB
  - In-order commit of stores also eliminates WAR and WAW hazards

- Issue #2: RAW dependences between loads and stores
  - Tougher because of non-static nature of long memory addresses
  - Solutions:
    - Total ordering of loads with respect to stores: simple but slow
    - Load bypassing: let loads bypass independent stores
    - Load forwarding: let stores forward results to dependent loads

- Speculative loads: assume loads independent from pending stores
  - If done carefully, can improve performance significantly
  - In any case, need to check and potentially correct mis-speculation
Review: Memory Disambiguation using Store Sets

- A mechanism to track load-store dependencies
  - Loads/stores assigned to sets as dependencies are discovered
  - When a load ready to issue, check if the last store in the set is still active
    - If yes, do not issue load; if no, speculatively issue the store

**Figure 6.1: Implementation of Store Sets Memory Dependence Prediction**

Loads and stores index into the SSIT to get their store set identifiers, which are used to access and update the LFST. The store inums that are found in the LFST indicate the memory dependence prediction.
Multiple LSUs & Non-blocking Caches

Implementation of 2-port cache?

What happens when a load misses?
Non-blocking or Lockup Free Caches

• Idea
  – Allow for hits while serving a miss (hit-under-miss)
  – Allow for more than one outstanding miss (miss-under-miss)

• When does it make sense (for L1, L2, …)
  – When the processor can handle >1 pending load/store
    • This is the case with superscalar processors
  – When the cache serves >1 processor or other cache
  – When the lower level allows for multiple pending accesses
    • Multi-banked, split transaction busses, pipelining, …

• What is difficult about non-blocking caches:
  – Handling multiple misses at the time
  – Handling loads to pending misses
  – Handling stores to pending misses
Non-blocking Caches

- Stall CPU on miss
- Hit under miss
- Stall only when result needed
- Multiple out-standing misses
Miss Status Handling Register

• Keeps track of
  – Outstanding cache misses
  – Pending load & stores that refer to that cache block

• Fields of an MSHR
  – Valid bit
  – Cache block address
    • Must support associative search
  – Issued bit (1 if already request issued to memory)
  – For each pending load or store
    • Valid bit
    • Type (load/store) and format (byte/halfword/…)
    • Block offset
    • Destination register for load OR store buffer entry for stores
<table>
<thead>
<tr>
<th>Valid</th>
<th>Block Address</th>
<th>Issued</th>
<th>Valid</th>
<th>Type</th>
<th>Block Offset</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load/store 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/store 1</td>
</tr>
<tr>
<td>Load/store 2</td>
</tr>
<tr>
<td>Load/store 3</td>
</tr>
</tbody>
</table>
Non-block Caches: Operation

• On a cache miss:
  – Search MSHRs for pending access to same cache block
    • If yes, just allocate new load/store entry
  – (if no) Allocate free MSHR
    • Update block address and first load/store entry
  – If no MSHR or load/store entry free, stall

• When one word/sub-block for cache line become available
  – Check which load/stores are waiting for it
    • Forward data to LSU
    • Mark loads/store as invalid
  – Write word in the cache

• When last word for cache line is available
  – Mark MSHR as invalid
VLIW and Static ILP
The Paradox of Superscalar Processing

• Compiler
  – Analyze sequential program to identify independent instructions
  – Produce sequential schedule with dependent instructions spaced apart
  – Map variables to a small set of registers by maximizing reuse

• Superscalar compiler
  – Analyze sequential schedule to identify independent instructions
  – Schedule instructions for parallel execution
  – Remap small register set into a large register set

• Idea behind VLIW
  – Design ISAs, compiler, and hardware the work synergistically for ILP
HW/SW DesignSpace for ILP

Compiler

Front end & Optimizer

Determine Depend.

Determine Independ.

Bind Resources

Hardware

Sequential (Superscalar)

Dependence Architecture

(Dataflow)

Independence Architecture

VLIW

Independence Architecture

(Attached Array Processor)

Determine Depend.

Determine Independ.

Bind Resources

Execute

[B. Rau & J. Fisher, 1993]
VLIW: Very Long Instruction Word Instruction Sets

- Long instruction words (or packets or bundles)
  - Each word contains multiple operations
- No data dependencies between operations in a word (parallelism)
  - No need for RAW checks
- Each operation slot corresponds to specific functional unit
  - Operation latencies are typically fixed
- Words spaced apart statically (using nops)
  - All operations are ready to execute
VLIW Implications

- The (optimizing) compiler must
  - Guarantee all operations within a long word are independent
    - Use nops when needed
  - Guarantee spacing between long words to avoid hazards
    - Use nops when needed

- But hardware is simple (few dynamic decisions)
  - No need for dependency checks within a word
  - No need for scheduling and issue hardware (instruction window)
  - Simple logic for functional unit assignment

- World length: 2 to 10ns of instructions, 64 to 1,000 bits
Early VLIW Models

- Josh Fisher proposed the first VLIW machine at Yale (1983)
  - Fisher’s Trace Scheduling algorithm for microcode compaction could exploit more ILP than any existing processor could provide at the time

- The ELI-512 provide massive resources to a single instruction stream
  - 16 processing clusters with multiple FUs per cluster
  - partial crossbar interconnect
  - multiple memory banks
  - attached processor – no I/O, no operating system

- Later VLIW models became increasingly more regular
  - Compiler complexity was a greater issue than originally envisioned
Ideal VLIW Design

Global Multi-Ported Register File

Functional Unit

Functional Unit

Functional Unit

Functional Unit

Instruction Cache

Sequencer

Condition Codes
Realistic VLIW Design using Clustering

Multi-Ported Register File

FAdd (1 cycle)

FMul 4 cyc pipe

FMul 4 cyc unpipe

FDiv 16 cycle

Instruction Memory

Sequencer

Condition Codes

EE382A – Autumn 2009 Lecture 18 Christos Kozyrakis
Clustered VLIW Architecture

- HW divided in clusters
  - Cluster: partition of registers and a few FUs
  - Full connectivity, low latency within a cluster
  - Explicit instructions to move data between clusters

- Advantage
  - Limit complexity of register file partitions

- Disadvantage
  - Compiler responsible for moving data between clusters
  - And to hide the latency of the transfers

- Clustering is common in embedded VLIW processors
Strengths of VLIW Technology

• Parallelism can be exploited at the instruction level
  – Available in both vectorizable and sequential programs.

• Hardware is regular and straightforward
  – Most hardware is in the datapath performing useful computations.
  – Instruction issue costs scale approximately linearly
    *Potentially very high clock rate*

• Architecture is “Compiler Friendly”
  – Implementation is completely exposed - 0 layer of interpretation
  – Compile time information is easily propagated to run time.

• Run-time behavior is highly predictable
  – Allows real-time applications.
  – Greater potential for code optimization.
Weaknesses of VLIW Technology

• No object code compatibility between generations

• Program size is large (explicit NOPs)

• Compilers are extremely complex
  – Assembly code is almost impossible

• VLIW memory systems can be very complex
  – Simple memory systems may provide very low performance
  – Program controlled multi-layer, multi-banked memory

• Parallelism is underutilized for some algorithms
Enabling Technologies for VLIW

- VLIW architectures achieve high performance through the combination of a number of key enabling hardware and software technologies.
  - Optimizing schedulers (compiler)
  - Loop unrolling & software pipelining (compiler)
  - Static branch prediction (compiler)
  - Symbolic memory disambiguation (compiler)
  - Predicated execution (compiler and HW)
  - (Software) speculative execution (compiler and HW)
  - Program compression (mostly HW)
Loop Unrolling

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

• Reduce loop overhead
  – Increment induction variable
  – Loop condition test

• Enlarged basic block (and analysis scope)
  – Instruction-level parallelism
  – More common subexpression
  – Memory accesses (aggressive memory aliasing analysis)
Software Pipelining

i=0
while (i<99) {
    ;; a[i] = a[i]/10
    Rx = a[i]
    Ry = Rx / 10
    a[i] = Ry
    i++
}

i=0
while (i<99) {
    Rx = a[i]
    Ry = Rx / 10
    a[i] = Ry
    Rx = a[i+1]
    Ry = Rx / 10
    a[i+1] = Ry
    Rx = a[i+2]
    Ry = Rx / 10
    a[i+2] = Ry
    i = i + 3
}

i=0
while (i<99) {
    Rx = a[i]
    Ry = Rx / 10
    a[i] = Ry
    Rx = a[i+1]
    Ry = Rx / 10
    a[i+1] = Ry
    Rx = a[i+2]
    Ry = Rx / 10
    a[i+2] = Ry
    i = i + 3
}
Software Pipelining (continued)

```
i=0
while (i<99) {
    ;; a[ i ]=a[ i ]/10
    Rx = a[ i ]
    Ry = Rx / 10
    a[ i ] = Ry
    i++
}
```

```
i=0
while (i<99) {
    Ry=a[ 0 ] / 10
    Rx=a[ 1 ]
    Rx = a[ i ]
    Ry = Rx / 10
    a[ i ] = Ry
    Rx = a[ i+1 ]
    Ry = Rx / 10
    a[ i+1 ] = Ry
    Rx = a[ i+2 ]
    Ry = Rx / 10
    a[ i+2 ] = Ry
    i++
}
```

```
i=i+3
a[97]=Ry
a[98]=Rx / 10
```
Software Pipelining Overhead

- Software pipelined loop require wind-up and wind-down code
  - Wind-up code is the $N$ iterations of the body necessary to set up the assumed register contents of the pipelined code
  - Wind-down code is the $M$ iterations of the body necessary to finish the computations started in the pipelined code
  - For small iteration counts, the overhead may be significant
- Sometimes can overlap wind-down of one loop with wind-up of another
Loop Unrolling Vs Software Pipelining

- How do they compare?
Trace Scheduling [Josh Fisher]

• Generate multi-basic block traces based on profiling information
  – Find the most often executed control path

• List schedule a trace at a time
  – Optimize the execution of the trace (common case)
  – Fix any problem with off-trace paths as necessary (infrequently executed)

• Good for very biased and predictable branching behavior
  – But it works for both loops and non loops

• Trace scheduling engendered the VLIW architecture innovation and was implemented in the Multiflow TRACE compiler, which provided the basis for superscalar compilation techniques now being used by Intel, HP, and DEC
Trace Scheduling Example

```plaintext
B1
fdiv f1, f2, f3
fadd f4, f1, f5
beq r1, $0

B2
ld r2, 0(r3)
fadd r2, r2, 4
beq r2, $0

B3
ld r2, 4(r3)

B4
add r2, r2, 4
beq r2, $0

B5
fsub f2, f2, f6
st.d f2, 0(r8)

B6
fsub f2, f3, f7

B7
add r3, r3, 4
add r8, r8, 4

B8
fsub f2, f2, f6
st.d f2, 0(r8)

B9
add r3, r3, 4
add r8, r8, 4
```

9 stalls

r2 and f2 not live out

1 stall
Predicated Execution

- Predicated Execution removes branches by *conditionally* executing operations.
- Removing branches combines multiple basic blocks into larger basic blocks.
- Branch related stalls are eliminated.
- Additional opportunities for scheduling optimizations appear.

Example:

```c
if (x<y) then
  z+=x;
else
  z+=y;

x=i+j;
y=k+m;
w=x+y;
```

Traditional Branching

```c
cc=(x<y)
branch (cc)

z+=y
z+=x

x=i+j
y=k+m
w=x+y
```

Predicated Execution

```c
cc=(x<y)
if (cc ) z+=x
if (cc ) z+=y

x=i+j
y=k+m
w=x+y
```
VLIW Today

- Servers: Intel IA-64 architecture
  - EPIC ISA (explicitly parallel instruction computing)
  - Chips Merced, Madison, Montecito, Tukwilla, ...
  - Questionable success compared to superscalars

- Embedded: TI, NXP, ST, ...
  - Very successful in low-end and high-end embedded SOCs
  - Rational: good performance/power, no need for binary compatibility
  - Large variety of ISAs, designs, optimizations points

- Interesting VLIWs of the recent past
  - Transmeta Crusoe (x86 on VLIW)
IA-64 vs. Classic VLIW

• Similarities:
  – Compiler generated wide instructions with ILP encoded in the binary
  – Static detection of dependencies
  – Large number of architected registers

• Differences:
  – Instructions in a bundle can have dependencies
  – Hardware interlocks between dependent instructions
  – Accommodates varying number of functional units and latencies
  – Allows dynamic scheduling and functional unit binding
    Static scheduling are “suggestive” rather than absolute
  – Code compatibility across generations
    but software won’t run at top speed until it is recompiled so “shrink-wrap binary” might need to include multiple builds
IA-64 Architecture

- 128 general-purpose registers
- 128 floating-point registers
- Arbitrary number of functional units
- Arbitrary latencies on the functional units
- Arbitrary number of memory ports
- Arbitrary implementation of the memory hierarchy

*Needs retargetable compiler and recompilation to achieve maximum program performance on different IA-64 implementations*
IA-64 Instruction Format

• IA-64 “Bundle”
  – Total of 128 bits
  – Contains three IA-64 instructions (aka syllables)
  – Template bits in each bundle specify dependencies both within a bundle as well as between sequential bundles
  – A collection of independent bundles forms a “group”
    A more efficient and flexible way to encode ILP then a fixed VLIW format

• IA-64 Instruction
  – Fixed-length 40 bits long
  – Contains three 7-bit register specifiers
  – Contains a 6-bit field for specifying one of the 64 one-bit predicate registers
Interesting Features of IA64

- Predicated execution
- Speculative, non-faulting Load instruction
- Software-assisted branch prediction
- Register stack
- Rotating register frame
- Software-assisted memory hierarchy
Predicated Execution

- Each instruction can be separately predicated
- 64 one-bit predicate registers
  - Each instruction carries a 6-bit predicate field
- An instruction is effectively a NOP if its predicate is false
- Assumes IA-64 processors have lots of spare resources
  - Converts control flow into dataflow
Speculative, Non-Faulting Load

- \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)
Speculative, Non-Faulting Load

- 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
Speculative “Advanced” Load

- *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
Using Speculative Load Results

- Inst 1
- Inst 2

potential aliasing

ld r1=[x]
use=r1

ld.a r1=[x]
inst 1
inst 2
use=r1

chk.a r1

ld r1=[a]
use=r1
Branch Prediction

- Static branch hints can be encoded with every branch
  - taken vs. not-taken
  - whether to allocate an entry in the dynamic BP hardware
- SW and HW has joint control of BP hardware
  - “brp” (branch prediction) instruction can be issued ahead of the actual branch to preset the contents of BPT and BTAC
    - Itanium uses a 512-entry 2-level BPT and 64-entry BTAC
- TAR (Target Address Register)
  - a small, fully-associative BTAC-like structure
  - contents are controlled entirely by a “prepare-to-branch” inst.
  - a hit in TAR overrides all other predictions
- RSB (Return Address Stack)
  - Procedure return addr is pushed (or popped) when a procedure is called (or when it returns)
  - Predicts nPC when executing register-indirect branches
Register Renaming

- 128 general purpose physical integer registers
- Register names R0 to R31 are static and refer to the first 32 physical GPRs
- Register names R32 to R127 are known as “rotating registers” and are renamed onto the remaining 96 physical registers by an offset
- Remapping wraps around the rotating registers such that when offset is non-zero, physical location of R127 is just below R32
Register Stack for Procedure Calls

• On a procedure call, the rename offset is bumped to the beginning of output argument registers
• Callee can then allocate its own working frame (up to 96 regs)
• If there isn’t enough free regs to be allocated, HW automatically frees up space by spilling life contents not in the current frame to memory

Register stack appears infinite to SW
Software-Assisted Memory Hierarchies

- ISA provides for separate storages for “temporal” vs “non-temporal” data, each with its own multiple level of hierarchies.
- Load and Store instructions can give hints about where cached copies should be held after a cache miss.
Latest Itanium Core

- 6 wide instruction fetch and issue
  - 6 wide integer, 2 wide FP, 4 wide ld/st, 3 wide branch
  - 1 cycle L1 data cache
  - 8-stage pipeline
  - 2-threads

- Memory hierarchy
  - Separate L1I, L1D, L2I, and L2D
  - 6MB L3
  - In 4-core Tuckwill: 30MB L3!