Class Overview

- This class is NOT about:
  - Programming languages
  - Parsing, syntax checking
  - Handling advanced language features – virtual functions, ...
  - Debugging
  - Simulation

- Focus is on compiler backend
  - Mapping applications to processor hardware
  - Work at assembly-code level (after “code generation”)
  - Retargetability – work for multiple platforms (not hard coded)
  - Primary goal is speed/efficiency
    - How to make the application run fast
    - Use less memory (text, data)

Background You Should Have

1. Programming
   - Good Java programmer (essential)
   - Linux, gcc, emacs
   - Compiler system not ported to Windows or Mac

2. Computer architecture
   - COS 471 or competency is sufficient
   - Know caches, pipelining, function units, registers, virtual memory, branches, branch prediction, assembly code

3. Compilers
   - I will assume you know nothing about the backend
   - Front-end, Grammars, Parsers, Code Generation (core COS 320 material) is not strictly necessary, but is important

4. Powerpoint
   - You will have to make at least one presentation in this class
What the Class Will be Like

- Classes will involve standard lectures
  - Some lectures will be prepared and presented by me alone
  - Some lectures will be jointly prepared and presented
  - Some classes will also involve discussion of a paper

- Roughly three canned homework assignments

- One final project

What the Class Will be Like

- Learning compilers
  - No memorizing definitions, terms, formulas, algorithms, etc.
  - Learn by doing – Writing code
  - Substantial amount of programming
    - Beware the leaning curve of the Velocity Compiler
  - Reasonable amount of reading

- Classroom
  - Attendance – You should be here
  - Discussion important
    - Work out examples, go over homework, etc.
    - Each of you will teach some advanced material to the rest of us
  - Essential to stay caught up (most material builds on prior material)
Course Grading

- Most students will get A’s
- Slackers will be obvious and will suffer
- If you want a blow off class, consider something else

Grade
- Homework – 33%
- Project – 33%
- Class participation and presentation – 33%

Homework

- Around 3 of these
  - Small/modest programming assignments
  - Design and implement something we discussed in class
  - Use the Velocity Compiler as a base
- Goals
  - Learn the important concepts
  - Learn the compiler infrastructure so you can do the project
- Working together is ok
  - Make sure you understand things or it will come back to bite you
  - Everyone must turn in their own assignment

The Project

- Design and implement an “interesting” compiler technique and demonstrate its usefulness
- Topic/scope/work
  - Up to 2 people per project
  - You will pick the topics (we have to agree)
  - You will have to
    - Read background material
    - Plan and design
    - Implement and debug
- Deliverables
  - Working implementation with full Java documentation
  - Project report – 1 page paper describing what you did/results
  - Presentation to the class
Class Participation

- Interaction and discussion is essential in a graduate class
  - Be here
  - Don't just stare at the wall
  - Be prepared to discuss the material
  - Have something useful to contribute

- Opportunities for participation
  - Useful comments, questions during lecture
  - Lecture Presentation
    - Put some effort into this – clear/illustrative presentation
  - Project Presentation
  - Feedback to me about the class

Contact Info

- Office: CS209 (also check CS004 – Liberty Lab)
- Email: august@cs.princeton.edu
- Office hours
  - After class for 30 minutes
  - Check CS209 and CS004

Importance of Compilers

- Hardware people have to understand compilers
  - No attention to compilers -> bad processor design

- COS320 material is not what real compiler people do
  - Parsing, syntax checking, etc. – a standard, mature field

- This class material is where all the action is at
  - How to make code run fast (approach hand coding)
  - How to reduce power
  - How to increase reliability
  - How to reduce code size
  - How to make use of unusual architectural features
  - How to design better processors
  - How to design customized processors
Computer Architecture

- Best way to start a compiler class is to talk about hardware!!
- Our target processor for this class is a VLIW/EPIC
  - Why?
- VLIW = Very Long Instruction Word
- EPIC = Explicitly Parallel Instruction Computing
  - IA-64: AKA Itanium I and II, Itanic, Merced, McKinley
  - Embedded processors –
    - All high-performance embedded CPUs are VLIW
    - TI-C6x, Philips Trimedia, Starcore, ...

The VLIW/EPIC Philosophy

- Compiler creates complete plan of run-time execution (POE)
  - At what time and using what resource
  - POE communicated to hardware via the instruction set
  - Processor obediently follows POE (if it can...)
  - No dynamic scheduling, out of order execution (these second guess the compiler's plan)
- Compiler allowed to play the statistics
  - Many types of info only available at run-time (branch directions, locations accessed via pointers, memory latencies)
  - Traditionally, compilers must behave conservatively → handle all cases
  - EPIC allows the compiler to gamble
    - Use profiling to determine the olds
    - Employ a mechanism to clean up the mess
- Expose microarchitecture to the compiler
  - memory system, branch execution

Defining Feature I – MultiOp Instructions

- Superscalar
  - Operations are sequential
  - Hardware figures out resource assignment, time of execution
- MultiOp Instruction
  - Set of independent operations that are to be issued simultaneously (no sequential notion within a MultiOp)
  - 1 instruction issued every cycle – provides notion of time
  - Resource assignment indicated by position in MultiOp
  - POE communicated to hardware via MultiOps
Defining Feature II - Exposed Latency

- Superscalar
  - Sequence of atomic operations
  - Sequential order defines semantics
    - Unit assumed latency (UAL)
  - Each conceptually finishes before the next one starts
- EPIC – non-atomic operations
  - Register reads/writes for 1 operation separated in time
  - Semantics determined by relative ordering of reads/writes
- Assumed latency (NUAL if > 1 for at least one op)
  - Contract between the compiler and hardware
  - Instruction issuance provides common notion of time

UAL vs NUAL example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Phase1 Operation</th>
<th>Phase2 Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r1 = load(r2)</td>
<td>v1 = load(r2)</td>
<td>r1 = v1</td>
</tr>
<tr>
<td>2</td>
<td>r1 = load(r3)</td>
<td>v2 = load(r3)</td>
<td>r1 = v2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r4 = mpy(r1, r5)</td>
<td>v3 = mpy(r1, r5)</td>
<td>r4 = v4</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r4 = add(r1, r6)</td>
<td>v4 = add(r1, r6)</td>
<td>r4 = v4</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r7 = mpy(r4, r9)</td>
<td>v5 = mpy(r4, r9)</td>
<td>r4 = v3</td>
</tr>
<tr>
<td>10</td>
<td>r7 = add(r7, r8)</td>
<td>v6 = add(r7, r8)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- NUAL

Assume load = 4 cycles, add = 1, mpy = 3

Other Architectural Features of VLIW/EPIC

- Additional features to support VLIW/EPIC philosophy
  - Create more efficient POEs
  - Expose the microarchitecture
  - Play the odds
- Register structure
- Branch architecture
- Data/Control speculation
- Memory hierarchy management
- Predicated execution
Register Structure

- Superscalar
  - Small number of architectural registers
  - Rename using large pool of physical registers during execution

- EPIC
  - Compiler responsible for all resource allocation including registers
  - Rename at compile time – large pool of architectural regs needed
  - Static renaming
    - Modify operands explicitly
  - Dynamic renaming
    - Operands not explicitly modified
    - Is this feature lost? NO!

Rotating Registers

- Overlap loop iterations
  - How do you prevent register overwrite in later iterations?
  - Compiler-controlled dynamic register renaming

- Rotating registers
  - Each iteration writes to r13
  - But this gets mapped to a different physical register
  - Block of consecutive regs allocated for each reg in loop corresponding to number of iterations it is needed

  \[ \text{actual reg} = (\text{reg} + \text{RRB}) \mod \text{NumRegs} \]

  At end of each iteration, RRB--

Branch Architecture

- Branch actions
  - Branch condition computed
  - Target address formed
  - Instructions fetched from taken, fall-through or both paths
  - Branch itself executes
  - After the branch, target of the branch is decoded/executed

- Superscalar processors use hardware to hide the latency of all the actions
  - Icache prefetching
  - Branch prediction – Guess outcome of branch
  - Dynamic scheduling – overlap other instructions with branch
  - Reorder buffer – Squash when wrong
EPIC Branches

- Make each action visible with an architectural latency
  - No stalls
  - No prediction necessary (though sometimes still used)
- Branch separated into 3 distinct operations
  1. Prepare to branch – compute target address, prefetch instructions from likely target
     - Executed well in advance of branch
  2. Compute branch condition – comparison operation
  3. Branch itself
- Branches with latency > 1, have delay slots
  - Must be filled with operations that execute regardless of the direction of the branch

VLIW/EPIC Advantages and Disadvantages

- Advantages
  - No implicit run-time dependence checks against previously or simultaneously issued operations
  - No run-time scheduling decisions
  - No register renaming
  - Rely on the compiler to do all the work
  - SIMPLER hardware, more effective (larger scope!)
- Disadvantages
  - Little tolerance for different or variable latencies
  - Little tolerance for a difference in the set of function units
  - Poor object code compatibility
  - More complex compiler

What About Bob?

- Bob doesn’t care about VLIWs
- He wants to compile for superscalars/RISCs?
  - All the basic compiler analyses and transformations are the same for all processor types
    - They were originally developed for RISCs
    - Delay slots are EPIC!
    - Scheduling to avoid hazards
  - Superscalar compilers work by pretending the processor is a VLIW
    - But must worry about hardware undoing what the compiler did
    - Other resources to worry about (reorder buffer, reserv stations, etc.)
    - Not all hardware features available
Take Away

- We will work with a variety of VLIW/EPIC machine models
- Decisions will always be made by the compiler unless otherwise noted

- Assignments involving performance will have a machine definition
- Machine for assignments will
  - Have no variable latencies or other architectural unpredictables
  - Only the program is unpredictable
- Matt will take responsibility for writing a scheduler and profiler to report performance (see board)

Going Forward

- Thursday
  - Control Flow Analysis I
  - HW #1 out (due February 19)

- I am traveling for the next two weeks

- Week II
  - Tuesday - Spyros: Velocity Compiler Overview
  - Thursday – Spyros: Advanced Control Flow

- Week III
  - Tuesday – Spyros: Region-based compilation
  - Thursday – HW #1 Due, NO LECTURE! (Go Skiing!)