Today's Plan

Division of Labor

- RISC vs. CISC
- Exposing parallelism

Hidden vs. Exposed Technology: RISC vs. CISC

CISC: Complex Instruction Set Computers

- VAX, Intel x86, . . .
- Instruction density was important, so architects were motivated to provide many features, i.e., lots of instructions

What do the following VAX instructions do?

- MNEGF Move negated floating point
- FFS Find first set bit
- INSQHI Insert at head of queue, interlocked
- POLYD Polynomial evaluation

The VAX had over 300 instructions

- Variable length
- Multiple complex addressing modes
  - eg. Load indirect off of an offset

Problems with CISC Architectures

Change in context

- Fewer hand-coded assembly programs
- More compiled code
- Compilers unable to effectively use complex instructions
- Large memories decrease the benefit of compact code

Collapsing under its own weight

- Increasingly complex control logic $\Rightarrow$ increasing use of microcode
- Microcode is flexible: essentially "interpreted control unit"
  - Easier to implement than hardwired control
  - Slower than hardwired control

Rise of RISC

RISC: Reduced Instruction Set Computers

- Fixed-length instructions
- Simple addressing modes
- Load/Store architecture
  - Only Loads and Stores access memory

Advantages of RISC?

Advantages of RISC Architectures

Regularity and simplicity provide implementation advantages

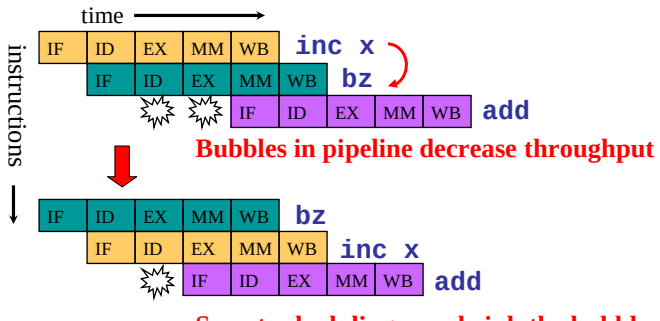
- Less complex control logic
- Faster clock rates
- Easier to pipeline operations
  - Easier to schedule multiple concurrent operations
  - Recall pipelined processors. . .

Scheduling to Improve Performance

Code Fragment

```
inc x
bz $r1, label        // if $r1==0, branch to label
add $r2,$r3,$r4
```

Pipeline Picture



Advantages of RISC Architectures

Compilation advantages

- Easier to compile for
- Simpler optimization model
  - No variable length instructions
  - Long latency instructions (loads, stores, branches) are exposed
  - Can re-order code to hide the latency of loads, stores, and branches

Don't pay for what you don't need

Lessons from RISC vs. CISC

If you want parallelism

- Design it into the language, in this case, the ISA

If you want effective use of your system

- Expose costs
  - Load/Store architecture exposes costs
  - Complex instructions hide costs

These lessons apply to languages and programming models as well

Today's Plan

Division of Labor

- RISC vs. CISC
- Exposing parallelism

Parallelism and the System Stack

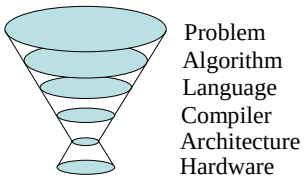
Superscalar view of the world

- The ISA is implicitly sequential
- The hardware then dynamically figures out what can execute in parallel

The problem

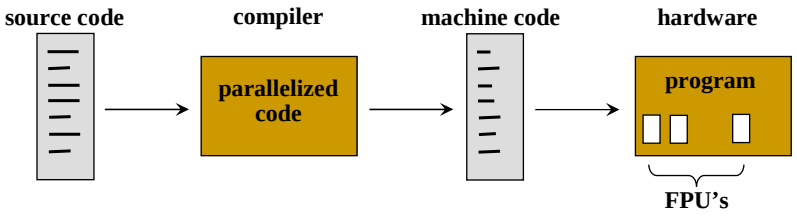
- Each level of the system reduces the amount of available parallelism

Moore's Funnel*



* [Chuck Moore, c 2003]

Implicitly Sequential Instruction Stream

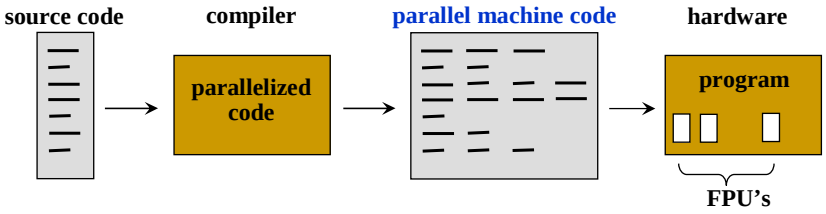


Problems

- Compilers can expose parallelism
- Compilers must eventually emit linear code
- Hardware must then re-analyze code to perform OoO execution
  - Hardware loses information available to the compiler
  - Compiler and hardware can only communicate through the sequential stream of instructions, so hardware does redundant work

How can we solve this problem?

Explicitly Parallel Instruction Stream



A solution

- Hardware does not need to re-analyze code to detect dependences
- Hardware does not perform OoO execution

VLIW: Very Long Instruction Word

- Each instruction controls multiple functional units
- Each instruction is explicitly parallel

VLIW

Basic idea

- Each instruction controls multiple functional units
- Rely on compilers to perform scheduling and to identify parallelism
- Simplified hardware implementations

Benefits

- Compiler can look at a larger window of instructions than hardware
- Can improve the scheduler even after a chip has been fabricated

Problems

- Slow compilation times
- No binary compatibility
  - Code is implementation-specific
- Difficult for compilers to deal with aliasing and long latencies

VLIW and IA-64

VLIW

- Big in the embedded market
  - Binary compatibility is less of an issue
- An old idea
  - Horizontal microcode
  - Multiflow (1980's)
  - Intel i860 (early 1990's)

Terminology

- EPIC: Explicitly Parallel Instruction Computer
  - New twist on VLIW
  - Don't make code implementation-specific
- IA-64 is Intel's EPIC instruction set
- Itanium was the "first" IA64 implementation

Explicitly Parallel Instruction Sets: IA-64

IA-64 Design Philosophy

- Break the model of implicitly sequential execution
  - Use template bits to specify instructions that can execute in parallel
  - Issue these independent instructions to the FPU's in any order
  - (Templates will cause some increase in code size)
- The hardware can then grab large chunks of instructions and simply feed them to the functional units
  - Hardware does not spend a lot of time figuring out order of execution; hence, simplified hardware control
  - Statically scheduled code
- Hardware can then provide a larger number of registers
  - 128 (about 4 times more than current microprocessors)
  - Number of registers fixed by the architecture, but the number of functional units is not

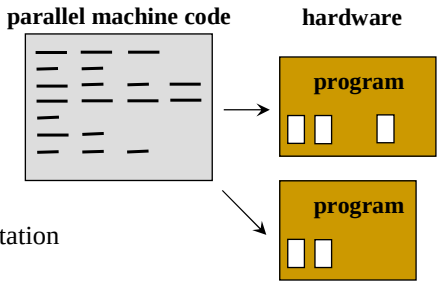
IA-64

A return to hardware "simplicity"

- Revisit the ideas of VLIW
- Simplify the hardware to make it faster
- Spend larger percentage of cycles doing actual work
- Spend larger percentage of hardware on registers, caches, and FPU's
- Use larger number of registers to support more parallelism

Engineering goal

- Produce an "inherently scalable architecture"
- Design an architecture—an ISA—for which there can be many implementations (lBM/360)
- This flexibility allows the implementation to change for "years to come"



Two Key Performance Bottlenecks

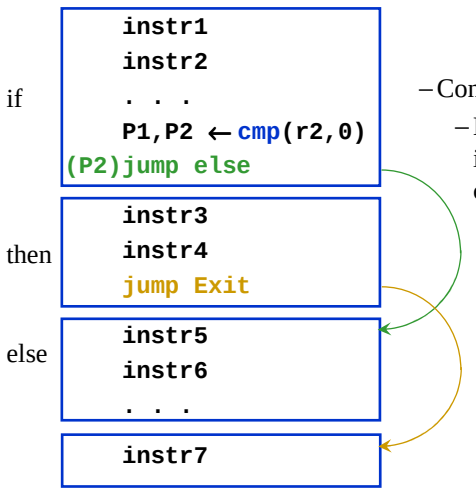
Branches

- Modern microprocessors perform good branch prediction
- But when they mispredict, the penalty is high and getting higher
  - Penalties increase as we increase pipeline depths
- Estimates: 20-30% of performance goes to branch mispredictions [Intel98]
- Branches also lead to small basic blocks, which restrict latency hiding opportunities

Memory latency

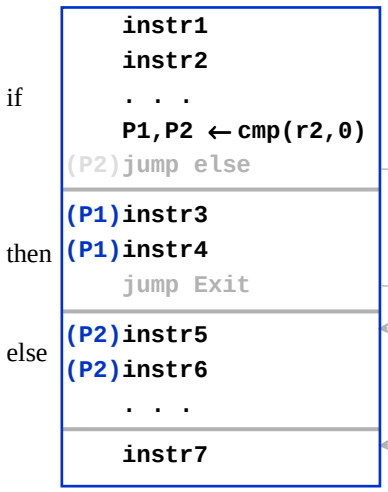
- CPU speed doubles every 18 months (60% annual increase)
- Memory speed increase about 5% per year

Branches Limit Performance



- Control dependences inhibit parallelism
  - Don't know whether to execute instr3 or instr5 until the cmp is completed

Predicated Execution



This is called *if-conversion*

Idea

- Add a predicate flag to each instruction
  - If predicate is true, the instruction is executed
  - If predicate is false, the instruction is not executed
  - Predicates are simply bits in a register
- Converts control flow into data flow
  - Exposes parallelism
  - With predicate flags, instr3 – instr7 can all be fetched in parallel

Benefits?

- Fewer branches (fewer mispredictions)
- Larger basic blocks
- More parallelism

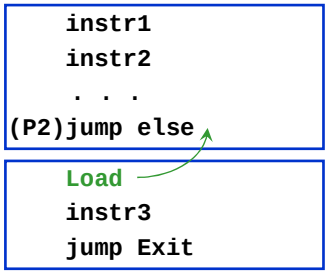
The Memory Latency Problem

Memory Latency

- Writes can be done out of order and can be buffered
- Loads are the problem: processor must wait for loads to complete before using the loaded value
- Standard latency-hiding trick: issue non-blocking load as early as possible to hide latency

The Problem

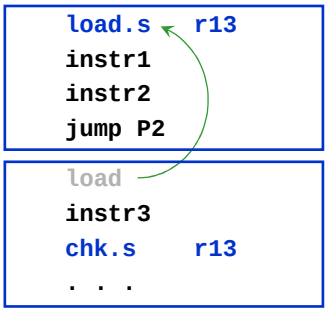
- Loads typically issued at beginning of basic block
- Can't move the Load outside the basic block
  - If the Load were to cause an exception when the basic block is not executed, then the early Load causes an erroneous exception



(Control) Speculative Loads

Split-phase operation

- Issue the load (load.s) as early as you wish
- Detect any exception and record it somewhere with the target of the load
- Can later check to see whether the load completed successfully: chk.s



Benefits?

- More freedom to move code– can now move Loads above branches as long as the check is in the original basic block
- Complication: What happens if chk.s is issued without a corresponding load.s?
  - This is clearly an error, so we need to be careful about where we move the load.s