Instruction Set Architecture

IA-64 and Compiler Support for Computer Architectures

1

Modern RISC processors

- Complexity has nonetheless increased significantly
- Superscalar execution (where CPU has multiple functional units of the same type e.g. two add units) require complex circuitry to control scheduling of operations
- What if we could remove the scheduling complexity by using a smart compiler…?

2

EPIC and IA-64

3

VLIW & EPIC

- VLIW – very long instruction word
- Idea: pack a number of *non-interdependent* operations into one long instruction
- Strong emphasis on ***compilers*** to schedule instructions
- Natural successor to RISC – designed to avoid the need for complex scheduling in RISC designs
- Example: IA-64

3 instructions scheduled into one long instruction word

IA- 64: The Itanium Processor

- A radical departure from the traditional paradigms.
- Intel and Hewlett-Packard Co. designed a new architecture, IA-64, that they expected to be much more effective at executing instructions in parallel
- IA-64 is brand new ISA, derived from EPIC (Explicitly Parallel Instruction Computing)

5

IA - 64

- 64-bit ISA
- Instructions are scheduled by the compiler, not by the hardware
- Much of the logic that groups, schedules, and tracks instructions is not needed thus simplifying the circuitry and promising to improve performance

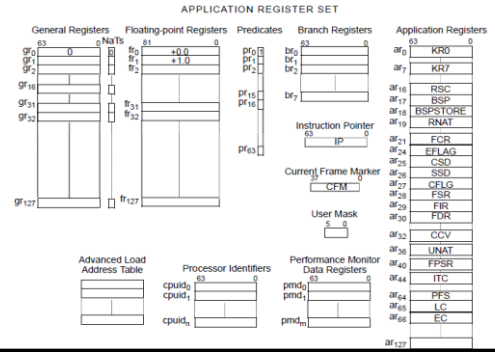
6

Intel IA-64

- Massive resources
  - 128 GPRs (64-bit not including the NaT/Not a Thing bit)
  - 128 FPRs (82-bit)
  - 64 predicate registers
  - Also has *branch registers* for indirect branches
- Contrast to:
  - RISC: 32 int, 32 FP, handful of control regs
  - x86: 8 int, 8 fp, handful of control regs
    - x86-64 bumps this to 16, SSE adds 8/16 MM regs

7

IA-64 Registers

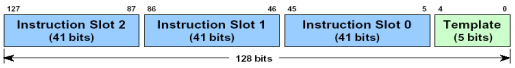


IA-64 Groups

- Compiler assembles *groups* of instructions
  - No register data dependencies between instructions in the same group
    - Memory dependence may exist
  - Compiler explicitly inserts "stops" to mark the end of a group
  - Group can be arbitrarily long

9

IA-64 Bundles



- Bundle == The "VLIW" Instruction
  - 5-bit template encoding
    - also encodes "stops"
  - Three 41-bit instructions
  - Bundles can be connected together and executed simultaneously
- 128 bits per bundle
  - average of 5.33 bytes per instruction
    - x86 only needs 3 bytes on average

10

Instruction Types

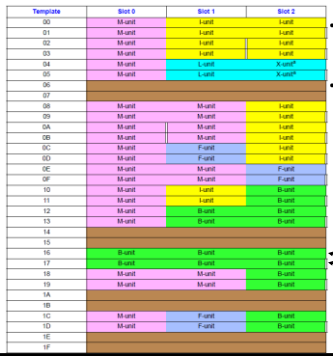
Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit/B-unit[a]

a. L+X Major Opcodes 0 - 7 execute on an I-unit. L+X Major Opcodes 8 - F execute on a B-unit.

- Instructions are divided into different types
  - the type determines which functional units the instruction operates on
  - templates are based on these types

11

Bundle Templates

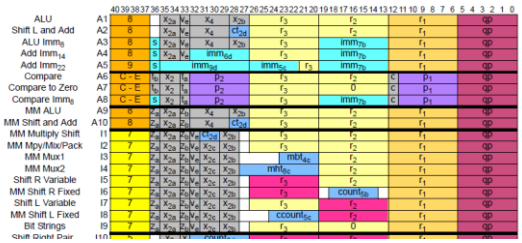


- Not all combinations of A, I, M, F, B, L and X are permitted
- Group "stops" are explicitly encoded as part of the template
  - can't stop just anywhere

12

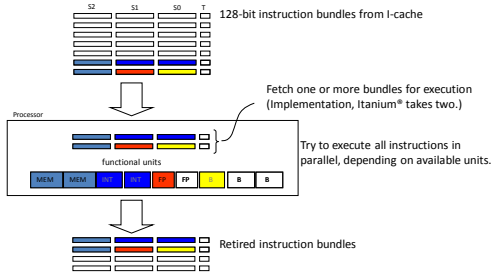
Individual Instruction Formats

- Fairly RISC-like like
  - easy to decode, fields tend to stay put



13

Explicitly Parallel Instruction Computing (EPIC)



14

The Role of Compilers

15

Compiler and ISA

- ISA decisions are no longer just for programming in assembly languages (AL) easily
- With HLLs, ISA is a compiler target today
- Performance of a computer will be significantly affected by compilers
- Understanding the compiler technology today is critical to designing and efficiently implementing an instruction set
- Architectural choices affect the code quality and the complexity of building a compiler for it

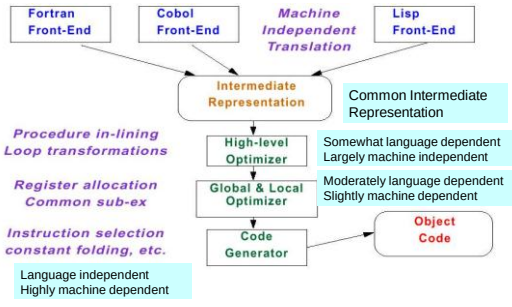
16

Goal of the Compiler

- Primary goal is correctness
- Second goal is speed of the object code
- Others:
  - Speed of the compilation
  - Ease of providing debug support
  - Inter-operability among languages
  - Flexibility of the implementation - languages may not change much but they do evolve - e. g. Fortran 66 ===> HPF

Make the common cases fast and the rare cases correct

Typical Modern Compiler Structure



18

Optimization Types

- High level - done at source code level
  - E.g., procedure called only once - so put it in-line and save CALL
- Local - done on a basic block (straight-line code)
  - Common sub-expressions produce same value
  - Constant propagation - replace constant valued variable with the constant - saves multiple variable accesses with same value
- Global - same as local but done across branches
  - Code motion - remove code from loops that compute same value on each pass and put it before the loop
  - Simplify or eliminate array addressing calculations in loop

19

Optimization Types (Cont.)

- Register allocation
  - Use graph coloring (graph theory) to allocate registers
    - NP-complete
    - Heuristic algorithm works best when there are at least 16 (and preferably more) registers
- Processor-dependent optimization
  - Strength reduction: replace multiply with shift and add sequence
  - Pipeline scheduling: reorder instructions to minimize pipeline stalls
  - Branch offset optimization: Reorder code to minimize branch offsets

20

Strength reduction

Example:

```
for (j = 0; j = n; ++j)
   A[j] = 2*j;

for (i = 0; 4*i <= n; ++i)
   A[4*i] = 0;
```

An optimizing compiler can replace multiplication by 4 by addition of 4.

21

Constant propagation

```
a:= 5;
 ...
// no change to a so far.
if (a > b)
   {
   . . .
   }
```

The statement `(a > b)` can be replaced by `(5 > b)`. This could free a register when the comparison is executed.

When applied systematically, constant propagation can improve the code significantly.

22

Register Allocation

- One the most important optimizations
- Based on graph coloring techniques
  - Construct graph based on the liveness of registers
    - Use a vertex to represent a variable,
    - Add an edge between two vertices if the two corresponding variables are live at the same time
  - If there are k registers, use k-coloring to allocate registers
  - Goal is to achieve 100% register allocation for all active variables.
  - Graph coloring works best when there are at least 16 general-purpose registers available for integers and more for floating-point variables.

23

Major Types of Optimizations and Examples

Optimization name	Explanation	Percentage of the total number of optimizing transforms
High-level	*At or near the source level; processor-independent*	
Procedure integration	Replace procedure call by procedure body	N.M.
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., $A = X$) with X	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	2%
Processor-dependent	*Depends on processor knowledge*	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

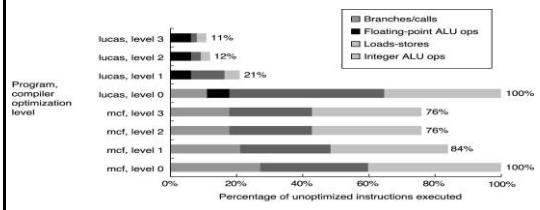
24

Practice Makes Perfection

- gcc optimization flags "-O1, -O2, -O3"
  - "-O0" turns off optimization
  - Example: gcc –O3 –o <out_file> <in_file>
- Examine the binary
  - objdump –D <executable or obj file>
  - View the output: "less <file>"
- Write a program, and see the difference with different optimization flags

25

Change in IC Due to Optimization



- Level 1: local optimizations, code scheduling, and local register allocation
- Level 2: global optimization, loop transformation (software pipelining), global register allocation
- Level 3: + procedure integration

26

How can Architects Help Compiler Writers

- Provide Regularity
  - Addressing modes, operations, and data types should be orthogonal (independent) of each other
    - Simplify code generation especially multi-pass
    - Counterexample: restrict what registers can be used for a certain class of instructions
- Provide primitives - not solutions
  - Special features that match an HLL construct are often un-usable
  - What works in one language may be detrimental to others

27

How can Architects Help Compiler Writers (Cont.)

- Simplify trade-offs among alternatives
  - How to write good code? What is a good code?
    - Metric: IC or code size (no longer true) →caches and pipeline…
  - Anything that makes code's performance easier to estimate
    - How many times a variable should be referenced before it is cheaper to load it into a register
- Provide instructions that bind the quantities known at compile time as constants
  - Don't hide compile time constants
    - Instructions which work off of something that the compiler thinks could be a run-time determined value hand-cuff the optimizer

28

Short Summary -- Compilers

- ISA has at least 16 GPRs (not counting FP registers) to simplify allocation of registers using graph coloring
- Orthogonality suggests all supported addressing modes apply to all instructions that transfer data
- Simplicity – understand that less is more in ISA design
  - Provide primitives instead of solutions
  - Simplify trade-offs between alternatives
  - Don't bind constants at runtime

29