Topics in computer architecture

Compilers and architecture

P.J. Drongowski
SandSoftwareSound.net

Copyright © 1990-2013 Paul J. Drongowski

Compilers and architecture

- Source: "Compilers and computer architecture," William A. Wulf, Computer, July 1981.

- Cost of hardware is falling dramatically
- Cost of software is rising rapidly
- Can hardware simplify the software task
- Goal: Better instruction sets to:
  - Simplify compilers
  - Improve size and speed of compiled programs

- Simplistic interpretation leads to mistaken inferences
  - Myth: Object code efficiency is unimportant
  - Aspirations grow faster than technology
  - Example: Graphical user interfaces
    - Must be "responsive"
    - Costly in both compute and memory resources

- Technological improvements
  - Hardware costs will continue to fall
  - Machine speed will increase
  - Memory will become more dense and less expensive

- User expectations will grow even faster
  - There will never be a memory or cycle surplus
  - Must increase the return from finite resources

Costs (compiler and architecture)

- Costs and benefits
  1 Designing (writing) compilers
  2 Designing the hardware architecture
  3 Designing the hardware implementation of the ISA
  4 Manufacturing the hardware
  5 Executing the compiler
  6 Executing the compiled programs

- Observations
  - All costs except 4 have increased
    - Hardware manufacturing costs have decreased
    - VLSI (chip level) design is expensive
  - All design activities are one time, non-recurring costs
    - Amortize over the number of units sold
    - Design a compiler-oriented architecture if
      - Compiler related costs (design, compile, execute)
      - Offsets cost of designing new architecture
    - More expensive to design hardware than software
  - Software lifetime exceeds lifetime of technology
  - Lifetime of architecture is longer than implementation
    - Architecture often caters to technology
    - Technology can pass an architecture by
  - Cost of compiling and execution
    - Not strictly comparable to other costs
    - Dollar cost can be assigned
    - Correct measure: Things that cannot be done as a Consequence of inefficiencies

Principles

- Regularity
  - Apply a feature in the same way everywhere
  - "Law of least astonishment"

- Orthogonality
  - Divide machine into a set of separate concerns
  - Define each feature in isolation of the others
  - Treat datatypes, addressing, operations independently

- Composability
  - Compose orthogonal, regular notions in arbitrary ways
  - Possible to use every addressing mode with every operator and every datatype

Case analysis

- Compiler performs an enormous case analysis
- Objective is to find best object code for source program
- Regularity, orthogonality, composability simplify analysis
- Every deviation is an *ad hoc* case to be considered
- Example: So-called "general" register machines
  - Implies that a register can be applied to any purpose
  - Exceptions (special cases)
    - Multiplicands in "even" registers
    - Double precision operands in even-odd pairs
    - Zero in indexing field implies no indexing (Makes the zeroth register unavailable for indexing)
    - Some operations are only register to register

Specific principles

- One versus all
  - Either precisely one way to do something
  - Or, all ways should be possible

- Provide primitives, not solutions
  - Synthesize solutions from primitives
  - Do not attempt to provide the solution itself

- Observations: one versus all
  - These extreme positions eliminate case analysis
  - Example: Conditional branching
    - EQUALITY and LESS THAN

    $\Rightarrow$ Only one way to generate each of six relations
    - Direct implementation of all six relations

    $\Rightarrow$ One obvious coding for each
    - Else, find cheapest code by commuting operands

- Observations: Primitives, not solutions
  - Avoid "semantic clash"
  - Do not put too much semantic content into instruction
  - Otherwise, use is limited to specific context
  - Example: FOR, CASE, PROCEDURE calls
    - Either support only one language well, or
    - So general that they are inefficient for special cases

More specific principles

- Address computations are paths
  - Addressing is not limited to arrays and records
  - Access involves following a path of arbitrary length
  - Path is known at compile time
  - Each setp along the path is:
    - An addition (index into an array or record)
    - An indirection (through a pointer)
  - Machines typically provide a large menu of modes

  ⇒ Requires exhaustive case analysis
  - Primitives, not solutions (again)
    - Different language constraints on procedures, tasks and exceptions
    - Example: C `switch` versus Pascal `case`

- Runtime environment support
  - Stack frames
  - Displays
  - Static and dynamic links
  - Exceptions
  - Processes

- Deviations
  - Only in implementation-independent fashion
  - Avoid technological anomalies
  - Look beyond current state of technology
  - Conjecture: Violations due to shortsighted view of costs

Regularity

- Operands treated symmetrically
  - Registers and memory interchangeable
  - Source and destination symmetric

- Operator - datatype regularity
  - Machines usually provide several datatypes
    - Different word sizes
    - Signed/unsigned integer, floating, address
  - Operators rarily treat all regularly
    - Operators for full-word integers but not bytes
    - Condition codes set inconsistently

- Beware of "arithmetic right shift"

- Immediate mode arithmetic
  - Frequently appearing constants: ± 1, 0
  - Special increment/decrement instructions
    - Sometimes useful only for forming addresses
    - Condition codes are not set in the same way
    - Carry not propagated beyond address size
    - Operate only on "index registers"

- Floating point instructions
  - Ideally an abstraction of real arithmetic
  - Sometimes not commutative or associative!

Orthogonality horrors

- Registers not treated alike
- Branching
  - Long and short branches
  - Displacement addressing may be unique to branches
- Addressing mode dependent operations
  - Sign-extension (not) done depending on destination
  - Even-oddness ⇒ long/short multiplication
- Different instructions for reg-reg, mem-mem, reg-mem, etc.

Composability

- Conversion
  - Relational operators
    - Relationals only affect control flow
    - HLL may allow assignment of Boolean value
  - Type coercion
  - Mismatch
    - Languages view type as property of data
    - Machines view type as property of operators
- Register allocation
  - Even - oddness of register use
    - $A \leftarrow B \times C$: Can be done "on the fly"
    - $A \leftarrow (B + D) \times C$: Must examine whole expression
  - Load / store motion
    - Move frequently used variables into registers
    - Eliminates load and store operations
    - Even - oddness may force analysis over basic block
  - Accumulator versus index register

One versus all (example)

- AND NOT provided instead of logical AND
- AND is commutative and associative
- AND NOT is neither
- Tedious analysis needed to:
  - Determine which operand to complement
  - Apply DeMorgan's laws to obtain optimal code

Primitives versus solutions

- "Semantic clash" between languages
  - Treatment of global data (e.g. COMMON, etc.)
  - Procedure parameter passing
  - FOR statements
  - Type conversions
- Machine design dilemma
  - Built-ins for one language cannot support others
  - Support for all will fail due to inefficiency
- Horrors
  - Support for only some parameter passing mechanisms
  - Certain loop models of initialization, test, recomputation
  - Address modes for certain stack frame or array layout
  - Case instructions that do (not) check boundary conditions
  - Case instructions that do (not) assume static bounds
  - Data structures different from common implementation
  - Elaborate string manipulation
- Complex instructions are usually composed of primitives

Addressing

- HLL permit arbitrary composition of:
  - Scalars
  - Arrays
  - Records
  - Pointers
- References can be quite complex
- Compiler must be able to handle the general case
- Further complications are due to:
  - Block structure
  - Recursive procedures
  - "By reference" parameter passing
- May require use of:
  - Indexing through a "display"
  - "Dope" (descriptor) information
  - Several levels of indirection
- Access is a path walking algorithm involving:
  - Indirection (following a pointer)
  - Computing a record element displacement
  - Indexing (by an array subscript)
  - Constraint checks on subscripts and nil pointers
- Machines tend to support an *ad hoc* collection of modes
  - No indirection at all!
  - Indexing/indirection in a fixed order
  - Type constraints on index multiplication
  - Limits on the size of displacement offset
- If more than one mode is available, choice is difficult

Environments

- Common language features
  - Recursive procedure invocation
  - Dynamic storage allocation
  - Process synchronization and communication
- Neglected areas
  - Uninitialized variables
    - Read before write
    - Set bad parity on uninitialized variables
  - Constraint checks
    - Subscript range checking
    - Case bounds checking
  - Exceptions
    - ON condition
    - Often violates hardware support for procedures
  - Debugging support
    - Force user to debug at low level (unpalatable)
    - Special debug mode (not much better)

Stacks

- Stack machines pose the same optimization problems
- Expression reordering
  - Reduces number of registers used
  - Also reduces stack depth
- Recompute or store is a difficult decision
  - Always advantageous on register machine to store
  - Must be offset by uses of value on stack