Instruction Set Architecture (ISA)

* Instruction set architecture of a machine fills the semantic gap between the user and the machine.

* ISA serves as the starting point for the design of a new machine or modification of an existing one.

* We will look at the main features of an instruction set architecture and learn how to design one.

Topics for Discussions

* Classifications of ISA
* Memory Addressing
* Operations in the Instruction Set
* Type and Size of Operands
* Encoding an Instruction Set
* Role of Compilers
* An Example ISA -- MIPS
  (in particular, MIPS64)

Assumed Prior Knowledge

* Instruction: Label Operator Operands
* Example: LOOP: MOVZ R1, R2
* Interpretation:
  Move the contents of register R1 (source operand) to R2 (destination operand) when the test flag is 0.
* Operands can be registers, memory address, or absolute values (immediate).
* Instruction cycle:
  Fetch, Decode, Execute, Store
* Basic memory and register addressing
* Registers Program Counter (PC) and Instruction Register (IR).

Classifying ISA

Stack Architecture:

Operands are implicit. They are on the top of the stack. For example, a binary operation pops the top two elements of the stack, applies the operation and pushes the result back on the stack.

Accumulator Architecture:

One of the operands is implicitly the accumulator. Usually one of the source operand and the destination operand is assumed to be the accumulator.

Classifying ISA (contd.)

General Purpose Register Architecture:

Operands are explicit: either memory operands or register operands. Any instruction can access memory. ISA design is orthogonal.

Load/Store Architecture:

Only load/store instructions can access memory. All other instructions use registers. Also referred to as register-register architecture.

Classifying ISA (contd.) - Sample Code

Stack	*Accumulator*	GP Register	*Load/Store*
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R1, B	Load R2, B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C, R3

Memory Addressing

* ISA design must define how memory addresses are interpreted and specified in the instructions.
* Most machines (ISAs) are byte addressed. That is, a byte (8 bits) is the smallest memory size that has a unique address. Other sizes are: half word, word, double word ...
* There are two different conventions for ordering bytes within a word: **Little Endian** and **Big Endian**.
* Little Endian: The byte with address XXX...00 is the least significant byte in the word.
  Big Endian: The byte with address XXX...00 is the most significant byte in the word.

Addressing Modes

* Addressing modes was one of the ways designers attempted to close the gap between HLL and the machine language. (By providing modes that a compiler can directly use to map many of the HLL features such as array indexing, base register addressing, position independent coding (PC-relative.)
* Examples: Indexed mode array access, auto increment and auto decrement to allow processing of array elements in sequence and so on.
* Some popular addressing modes: Register, Immediate, Displacement, register-indirect, indexed, absolute, memory-indirect, auto increment and auto decrement.

Choice of Addressing Modes

* Study the measurements related to the usage of the various addressing modes.

* For example, if the displacement mode is used, you may further study the distribution of how far the displacement is.

* Then, select the modes depending on their usage, and the specific attributes of the mode depending on experimental results with a select set of benchmarks.

Operations in the Instruction Set

* Data transfer instructions.
* Arithmetic and logic instructions.
* Instructions for control flow: conditional and unconditional branches, jumps, procedure calls and procedure returns.
* System calls.
* Floating point instructions.
* Decimal instructions.
* String instructions.
* Graphics instructions.

Behind the Scenes Operations

* **Condition codes** and condition register: Should this be set after every instruction or only when specified?

* For procedure calls, does the caller or called save **the state of computation**?

* **Position independence**: Most common way to specify branch destination is to supply a displacement that is added to the Program Counter (PC). Using PC-relative addressing permits the code to run independently of where it is loaded.

Type and Size of Operands

* Type: Integer, floating point, single precision float, double precision float, character.

* Floating point: IEEE 754 format.

* Support for decimal data: Direct decimal operations, packed decimal.

* byte, word, double word, quad word, etc.

* With SPEC92 suite of benchmarks, on 32-bit address machines, word is the size commonly used for integer computation and 64-bit (double) is common for floating point computation.

Encoding an Instruction Set

* Operator of the instruction is encoded into **opcode**.

* When there are large number of addressing modes and more than one memory operands are allowed in an instruction, each operand must have an **address specifier**.

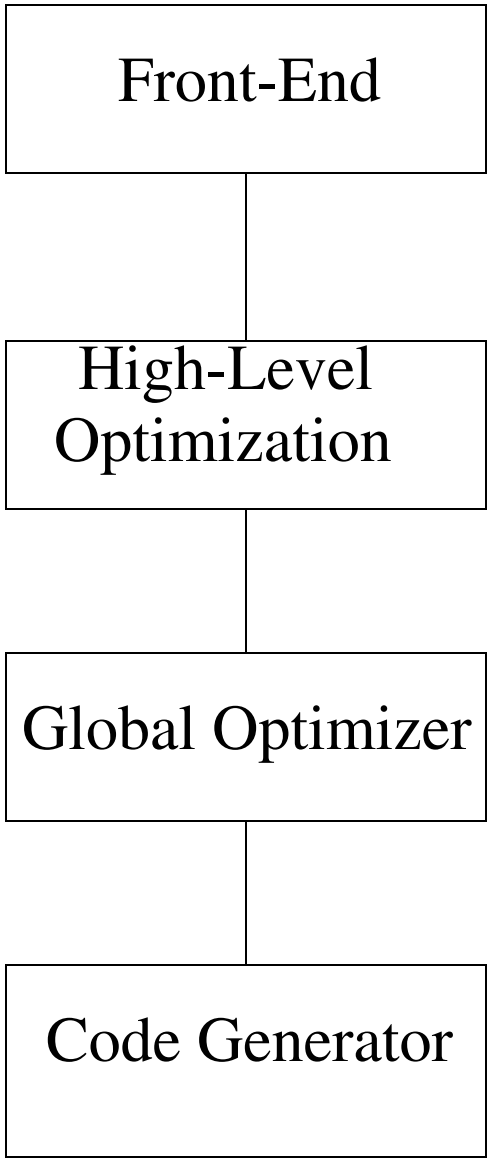
* Otherwise if it is like the load/store machine, the operands can be encoded along with the opcode itself.

* The number of addressing modes and number of registers have a significant effect on instruction length.

Variations in Instruction Encoding

* Variable:
  Can support any number of operands.
  Number of operands is a part of the opcode.
  Each operand has an address specifier.
  (e.g. VAX, 1-53 bytes)
* Fixed:
  Fixed format always has the same number of operands. How they are decoded may differ with opcode. (e.g. SPARC (4 bytes), MIPS, PowerPC)
* Hybrid:
  Multiple formats of fixed length. Format specified as a part of the opcode. (e.g. IBM 370, Intel 80X86 series (7 different formats of 1-5 bytes)

The Role of Compilers



Transform HLL to common intermediate form

Procedure inlining & loop transformations

Global and local opt. & register allocation

Machine dependent opt.

The Role of Compilers (contd.)

* Front-end:
  One per language. Once the source code is transformed into a common intermediate code, the rest of the phases are the same for any HLL.

* High-level optimization:
  Involves working on the source code.
  Example: online expansion of functions.

* Local optimization:
  Within a sequence of code.
  The straight line section of the code.

The Role of Compilers (contd.)

* Global optimization:
  External local optimization across branches and introduce a set of transformations aimed at optimizing loops (Loop Optimization).

* Register allocation is a very important function. Recent register allocation techniques are based on **graph coloring**.

* Machine-dependent optimizations exploit some architectural features.

Compiler Technology and ISA

* Today's compiler technology (in the form of optimization and others) helps in making such architectures as load/store a viable solution.

* As an ISA designer, one may help the compiler writer generate efficient code. What are the features in ISA that can help compilation?

ISA Features to Support Efficient Compilation

* Regularity:
  Three components of instruction set: operation, data types and addressing modes should be orthogonal. (independent)
  e.g. VAX's arithmetic operations are equally applicable to register or memory operands.

* Provide primitives and not solutions:
  Features that match a particular HLL feature are often not usable by other HLL or other statements.
  e.g. Index register to support arrays. Instead a general purpose register would serve the purpose and other situations well.

ISA Features ... (contd.)

* Simplify trade-off among alternatives:
  An HLL statement may be translated into several different equivalent sets of instructions.

  -- Which is better? The longer or shorter sequence? With cache and pipelining this tradeoff becomes complex.

  -- Example:
     In register-memory architecture, how many times a memory variable has been referenced before it can be moved into a register?

ISA Features ... (contd.)

* Provide instructions that bind quantities known at compile time as constants --
  if something is known at compile time, resolve it at compile time.

* Counter Example -- CALLS of VAX ISA
  CALLS is a procedure call in VAX's ISA. Even though the list of registers to be saved on transfer control to the procedure is known at compile-time, the register saving process is included in the semantics of CALLS! So cannot be substituted with instructions at compile time.

Summarizing the ISA Features

* How to decide the general class of ISA?

* How is the memory going to be addressed?

* What are the addressing modes?

* What are the allowed operations?

* The types of operands?

* How are instructions encoded?

* How to add features that help compiler generate efficient code?