

Instruction set architectures

- ISA metrics
- Basic ISA taxonomy
- Differentiating ISAs
 - Memory access
 - Big vs. little endian
 - Alignment
 - Addressing modes
 - Kinds of instructions
- Making life easier for compiler writers
- Example ISA: DLX

ISA metrics

- Instruction density: how much space per task?
 - Instruction count: how many instructions per task?
 - Instruction complexity
 - How much decoding is necessary?
 - How many “simple” operations per instruction?
 - Instruction length
 - Constant vs. variable-length
 - Average instruction length
- ⇒ Keep these metrics in mind when discussing individual ISAs

ISA taxonomy

- Accumulator
 - One operand is in the accumulator, the other in memory
 - Instructions move data between accumulator and memory
- Stack-based
 - All operands on the top of the stack
 - Instructions move data between top of stack and rest of memory
- These are less common ISAs for CPUs
 - Java Virtual Machine is stack-based
- More common today: General Purpose Register (GPR) ISAs

GPR ISAs

- Memory-memory (example: VAX)
 - May have as many as 3 operands in memory
 - Usually have relatively few registers (used to save memory references)
- Register-memory (example: 680x0)
 - One operand in a register, the other may be in memory
 - Usually has only two operands (register is both source & dest)
 - Generalization of accumulator ISA
- Load-store (example: PowerPC, MIPS, DLX)
 - Data moved explicitly between registers and memory
 - ALU operates on registers (usually 2 source & 1 destination)

More on GPR ISAs

- GPR ISAs are the most popular design today
 - Registers are much faster than memory (2 ns vs. 70ns)
 - Compilers can optimize register use for:
 - Holding intermediate values in calculations & addresses
 - Caching variable values
 - Passing parameters
- GPRs can be classified by
 - The number of ALU operands (2 or 3)
 - The number of operands in memory (0-3)

Accessing memory

- Endianness
 - Big-endian: most significant byte stored in first byte of word
 - Little-endian: least significant byte stored in first byte of word
 - No intrinsic advantage to either approach
 - Networks usually use big-endian for transmitting words
- Alignment: must n -byte objects be aligned to addresses evenly divisible by n ?
 - Advantage: more flexible for programs
 - Disadvantage: more complicated hardware
 - May be relaxed slightly, ie, 8-byte objects 4-byte aligned

Addressing modes

- Instructions specify operands using addressing modes
- Register: value is in a register (ADD R4, R5, R6)
- Immediate: value contained in the instruction (ADD R4, #4)
- Memory
 - Indirect / Displacement: address in a register (LW R1, 8(R4))
 - Indexed: add two registers to get address (LW R1, 8(R1+R4))
 - Modifiers include
 - Auto-increment/decrement: used for stacks (LW R1, (R2)+)
 - Scaling: address scaled by size of data
 - Multiple levels of indirection
 - Absolute/Direct: address contained directly in instruction

More addressing modes

- PC-relative addressing
 - Use current value of PC as the base rather than a register
 - Often used for branches and static variables
- Other addressing modes
 - Register update on PowerPC: put the effective address into the base register (implements auto-increment/decrement flexibly)
 - Implicit: top of stack or accumulator used “by default”
 - Memory deferred: multiple levels of indirection on a memory access

Implications of addressing modes

- More addressing modes can
 - Lower instruction count & increase code density
 - Increase implementation complexity
 - Increase CPI (maybe)
 - Reduce execution time (maybe)
- Fewer addressing modes can
 - Increase instruction count
 - Allow smaller, simpler (and easier to decode) ISA
 - Decrease CPI (maybe)
 - Reduce execution time (maybe)

Minimum set of addressing modes

- Register
 - Must have a way of accessing registers!
 - Not necessary for stack-based architectures
- Indirect
 - Need a way of accessing memory
 - May require that all addresses first be loaded into a register...
 - Displacement isn't much harder (often used)
 - 75% of displacements are <12 bits
 - 99% of displacements are <16 bits

Minimum set of addressing modes

- Immediate
 - Must be a way of getting constant values into the CPU
 - Could use constants in memory, but how would they get there?
 - How big must immediate values be?
 - Most values are less than 8 bits (50%+)
 - Addresses require large immediate values: combine two 16-bit values to make a 32-bit address
 - Usually use 8-16 bit immediates
- Other addressing modes could be useful, but are they worth the complexity?

Instruction set operations

- Required operations
 - Arithmetic/logical: ALU operations
 - Loads/stores: moving data between registers & memory
 - Control: branches, jumps, subroutines, traps
 - Optional (but very useful!)
 - Operating system support: traps (OS calls), VM support (TLB) & cache support
 - Floating point
 - Other operations (may be useful for some situations)
 - Graphics / vector operations (becoming more common)
 - Binary-coded decimal & string (becoming less common)
- ⇒ **Make the common case (ALU, load/store, control) fast!**

Control flow instructions

- Types of control flow instructions
 - Conditional branches (>80% of all control flow instructions)
 - Jumps (also unconditional branches)
 - Procedure calls & returns
- Addressing for control flow
 - PC-relative: allows position-independent (relocatable) code
 - Commonly used for branches & jumps
 - Indirect: address in register; used for
 - Procedure return in some architectures
 - Jump tables (switch/case statements) & virtual functions
 - Dynamic linking
 - Absolute: address in instruction; used for ROM calls

Conditional branches

- Displacement length
 - Most conditional branches go fewer than 10 instructions away
 - Fields of 8 bits (signed) should be enough for most situations!
- Branch condition
 - Compare register to 0/1: test condition and put result in register
 - Compare register to other values (register, immediate)
 - Condition codes
 - Comparison sets flags in the CPU
 - Can be slower and involves more state to keep track of
- Branch destinations
 - Non-loop branches usually forward (75% of branches, hard to predict)
 - Looping branches tend to be backward (usually taken)

Subroutines

- Transfer control and save some state
 - At a minimum, save return address
 - Save register state: in CALL instruction or done by compiler?
 - Caller save: calling routine saves registers it'll use after routine
 - Callee save: called routine saves registers it'll modify
- Return from subroutine
 - Restore original state
 - Jump to instruction after subroutine call
- Complex subroutine call / return can be very slow
 - Provide simple instructions & let compiler combine instructions to save the necessary state
 - Leaf subroutines may not even need to access memory to save state!

Fixed & variable length instructions

- Variable length instructions (common in CISC)
 - Compose instructions of “pieces”: operation type, operand specifiers
 - Pack more instructions into a fixed space (better code density)
 - Decoding is more difficult: must decode instruction to figure out where the next one starts
 - Instruction fetch must be able to handle unaligned accesses
- Fixed length instructions (common in RISC)
 - Operation & addressing modes fixed into opcode
 - Supports relatively few addressing modes: common in load/store
 - Decoding is much less complex
 - Prefetch many instructions ahead without decoding intervening instrs
 - Code is less dense: requires more memory for given functionality

Goal of ISA design: compilers

- 99%+ of computer code produced by compilers
 - Make an instruction set easy for a compiler to use
 - Make it possible (but not necessarily easy) for a human to read
- Compiler passes
 - Parsing / language front end
 - High-level optimization: inlining, loop unrolling
 - Global optimization: register allocation, subexpression elimination
 - Code generation: output the actual assembly or machine language
 - Instruction selection
 - Instruction reordering
 - Delay slot filling

Designing an ISA for compiler use

- Provide a regular instruction set
 - Three components of ISA (operations, data types, addressing modes) should be orthogonal: all operations work on all data types & addressing modes
 - General purpose registers (vs. special purpose)
 - Provide primitives, not full solutions
 - Complex instructions may not match language (C vs. Pascal strings)
 - Allow the compiler to build up its own code sequences
 - Simplify tradeoffs
 - Don't make the compiler writer choose from 20 options!
 - Allow constants to be specified at compile time
- ⇒ **KISS: KEEP IT SIMPLE, STUPID!**

DLX: example load/store ISA

- Skim the material on the DLX ISA
- Highlights include
 - Simple load/store architecture
 - Relatively large register set (32 GPRs)
 - Only load/store can access memory; all other instructions operate on registers
 - Addressing modes
 - Displacement (16-bit signed offset)
 - Immediate (16-bit signed or unsigned values)
 - Register
 - Fixed length instructions (32 bits)

Here be dragons...

- Don't design an ISA oriented towards a specific HLL
 - Attempts to reduce the semantic gap may result in a semantic clash!
 - Instruction mismatch is likely, in which special instructions do more work than is required for the frequent case.
- There is no such thing as a typical program.
 - Programs can vary significantly in how they use an instruction set
 - Many times it is meaningless to average frequency criteria over several programs (i.e. the mix of data transfer sizes)
- Avoid the temptation to put in lots of cool instructions
 - What's cool today may be useless tomorrow
 - You'll have to support for a *very* long time

Building the perfect ISA

- There's no such thing as a flawless ISA
 - Every ISA involves tradeoffs!
 - Doing one thing well means doing something else less well
 - The perfect ISA for one program isn't perfect for another program
 - Predicting technology 10+ years into the future is very difficult!
- Flawed ISAs can be successful (example: Intel x86 ISA)
 - Register architecture is messy (no GPRs)
 - Segmentation is somewhat messier than pure paging
 - Stack-based FPU isn't as efficient as register-based FPUs
- ISAs eventually die off, but it takes a while
 - Intel x86 ISA still going strong
 - Motorola 680x0 ISA runs PalmPilots, printers, and more!