# Instruction Set Architecture

## Introduction

- Instruction set architecture (ISA)
  - The portion of the architecture visible to the programmer and the compiler
  - Consists of the definition of instructions supported by the machine
  - Necessarily includes components that determine the instruction set e.g. number of registers, number of data types, types of registers, memory addressing techniques etc.
  - At the time of design different ISAs are compared based on simulation that measures different metrics
  - Possible metrics could be code size, complexity of design, backward compatibility, power consumption, and of course performance of benchmarks



## Three market sectors

- Desktop
  - Performance is very important (both integer and fp)
  - Power consumption is equally important
  - Code size is of little or no importance
  - Backward compatibility is very important for success
- Server
  - For databases and web services integer performance is much more important
  - For supercomputing floating-point performance is more important
- Embedded
  - Cost, power, code size are very important; floating-point performance is less important (depending on app.)



## DSP and media

- Real-time performance is important
  - Worst-case performance must meet real-time deadline
  - Heavily optimized hardware for small number of frequently used kernels e.g. FFT, convolution, filters
  - ISAs often include special instructions to exploit this hardware; kernels are provided by manufacturer in the form of hand-coded library that makes use of these instructions (less reliance on the compiler)
  - Notice the difference from desktop/server segment
  - Today most desktop processors try to support DSP and media processing instructions

# Classification

- Four major ISAs
  - Stack architecture: Push A; Push B; Add; Pop C
    - Implicit stack operands
  - Accumulator architecture: Load A; Add B; Store C
    - Implicit accumulator operand
  - Register–memory architecture: Load R1, A; Add R2, R1, B; Store R2, C
    - Explicit memory operand in ALU instruction
  - Register–register architecture: Load R1, A; Load R2, B; Add R3, R1, R2; Store R3, C
    - Uniform format for all ALU operations
    - Also known as load/store register architecture
- Basic distinction is type of internal storage accessible to ALU

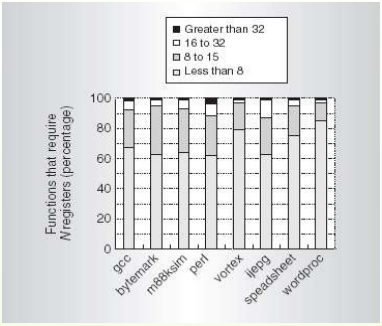


# Classification

- Popularity of load/store register ISA
  - Registers are faster than memory (so provide more general purpose registers)
  - Registers are easy to handle for the compiler compared to, say, stack: (A*B)–(B*C)–(A*D)
  - Intermediate values can be held in registers for fast access
  - Registers reduce memory traffic
  - Same type of ALU instructions take equal amount of time to execute: easier to pipeline and optimize (more later)
- Number of registers
  - Depends on how good the compiler is
  - Most compilers reserve some registers for parameter passing, for procedure return address, global memory pointer, etc



# Number of GPRs




For AMD Opteron (Reproduced from IEEE Micro)



# Classification

- Number of operands in ALU instructions can be 2 or 3
  - Some of them could be register operands while the rest are memory operands
  - Zero memory, 3 register: two sources, one destination; most popular register–register ISA (Alpha, MIPS, ARM, SPARC, PowerPC, ⋯)
  - One memory, one register: register–memory found in x86, Motorola 68k; one operand is both source and dest
  - 2 memory, zero register: memory–memory found in VAX
  - 3 memory, zero register: memory–memory found in VAX
  - Advantages and disadvantages? Instruction size, code density, ease of pipelining, execution time, number of bits for register encoding

# Memory addressing

- Accessing a value in memory requires two things
  - Starting address and length
  - Load/store register ISAs normally encode the length in the opcode i.e. they offer different instructions for different lengths (byte, half word (2 bytes), word (4 bytes), double word (8 bytes))
  - Access for size of x bytes normally needs to be aligned i.e. Address % x must be zero
    - Why?
  - On some computers it is possible to access least significant parts of a register e.g. x86; leaves upper portion unaffected



# Endianness

- Byte ordering within a word
  - Little endian machines place the byte with address x···xxx00 in the least significant position i.e. the little end (Alpha, Intel x86, VAX)
  - Big endian machines place the byte with address x···xxx00 in the most significant position i.e. the big end (MIPS, Sun UltraSPARC, Motorola 68k)
  - This ordering remains transparent to the programmer as long as he/she does not try to access the byte as well as the word starting at the same address
    - int x = 0x12345678; char *c = (char*)&x; print (*c);
  - Same rule for word ordering within a double word



# Addressing modes

for virtual addresses

- How to specify an address in an instruction (could be register, memory or immediate)
  - 10 major addressing modes (VAX had all)
    - Register: Add R4, R3 (assume two operands)
    - Immediate: Add R4, 3
    - Displacement: Add R4, 100(R1)
    - Register indirect: Add R4, (R1)
    - Indexed: Add R4, (R1+R2)
    - Direct or absolute: Add R4, (1000)
    - Memory indirect: Add R4, @(R3)
    - Autoincrement: Add R4, (R3)+
    - Autodecrement: Add R4, –(R3)
    - Scaled: Add R4, 100(R2)[R3]



# Addressing modes

- Need to choose the effective modes only
  - Implication on complexity, CPI, instruction count
  - Large number of addressing modes normally increase complexity, decrease instruction count (assuming the target applications and the compiler can exploit them), may increase CPI
  - Designers normally simulate the target benchmarks to see the relative usage of addressing modes
  - For example, on VAX, immediate and displacement modes are most heavily used by TeX, Spice and gcc benchmarks; Memory indirect, scaled, register indirect in addition to the above two cover almost all memory accesses; register indirect could really be displacement!
  - VAX designers argued that the frequency of usage depends on programming language and compiler

# Displacement mode

- How big a displacement?
  - Varies a lot
  - Zero displacement is most frequent
  - Most displacements are positive
  - Large displacements (requiring 14+ bits) are normally negative: need sign extension
  - Storage layout and hence the programming language may influence the displacement size



# Immediate mode

- Used for moving constants, arithmetic, comparison
  - Two major questions
    - Which instructions should support immediate mode?
    - How many bits should be devoted to immediates?
  - Load immediate (what is this?) and ALU immediate instructions are most frequent
  - Small immediate values are heavily used in arithmetic
  - Large immediate values are normally used for address constants e.g. some global offset



# DSP world

- Two special addressing modes are frequently seen
  - Modulo or circular addressing mode: autoincrement/autodecrement with automatic reset when end of buffer reached (useful for streaming data)
  - Bit reverse addressing mode: specially designed for FFT
  - Note that compilers may never be able to generate these addressing modes (because identifying such a situation is very hard); hence the need for hand-assembled library
  - Future may be different as DSP applications get larger and compilers become more important



# Summary: addressing

- Addressing modes should be simple
  - Displacement, immediate are popular; can emulate register indirect with displacement
  - Must match the ability of the compiler to use them (very important in desktop/server market)
  - Displacement should be 12–16 bits
  - Immediate should be 8–16 bits (in reality displacement and immediate fields use the same space in ISA encoding)

# Size of operands

- Specified in the instruction opcode
  - Byte, half word, word, double word
  - Characters are usually bytes
  - Integers can use half word, word, double word depending on what data types the programming language offers
  - Floating point numbers are normally expressed in IEEE 754 format: 32 bits (one word) for single precision, 64 bits (double word) for double precision
  - For business transaction, accuracy of decimal arithmetic may be important: packed decimal or binary coded decimal (BCD) supported by x86; consider the decimal number 0.1 and try expressing it in binary
  - Fixed point in GPUs and DSPs; separate integer part and fraction part; wide accumulator to avoid round-off error



1, 2, 4, 8 Byte

point posn fixed

m + n

4 bits all b/w 0 and 9

20.1

0010 0000 . 0001

BCD (avoids losing money)

int frac

Largest no. = $2^m - 1 + \sum_{i=1}^{n} \frac{1}{2^i}$

if mantissa & exponent m, n ⇒ very large range

Fixed pt ⇒ avoids errors ⇒ all ops on ints.
Mantissa Exp ⇒ big range.

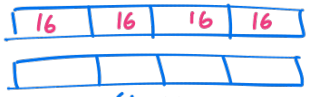
# Operations

- Simplest operations are used most frequently
  - 10 x86 instructions are sufficient to cover 96% of the entire SPECint92 suite
  - Commonly supported: arithmetic & logic, load/store, control transfer, system call, floating-point
  - Optionally supported: decimal arithmetic, string operations, graphics
  - Media and signal processors normally operate on narrow-width data; possible to execute multiple such operations in parallel (called Single Instruction Multiple Data operations)
  - DSPs normally support saturating arithmetic: on overflow taking an exception is out of question (real-time bound); so saturate to maximum value
  - DSPs use Multiply accumulate (MAC) operations; MACs/s is an important metric
    - a*b+c



Short vector ops

inner product




like adding 2 4D vectors in 2 reg addn

# Control transfer

- Four major types of instructions
  - Conditional branches (most frequent: ~20% of all ins.)
  - Unconditional jumps (direct and indirect)
  - Procedure call
  - Procedure return
- How to specify the target?
  - If compiler can figure out the target address it includes it in the instruction
  - Most frequent one is PC-relative (position independent, reduces linker burden)
  - Otherwise any addressing mode can be used; for procedure call and direct jumps the target is normally included in the instruction as a large constant



direct unc. goto label

if()

jmp addr

direct/indirect

always indirect uncondn

compiler may not know where to go back from where the fnc was called.

for condn branches PC relative possible in fewer bits.

# Control transfer

- Indirect jumps and procedure return
  - Not known at compile time
  - Any addressing mode (other than absolute) can be used to specify the target
  - Simplest one is register addressing i.e. place the target in a register
  - Indirect jumps appear when using switch/case blocks, virtual methods, function pointers, dynamically shared libraries; normally the target is loaded from memory into a register at run time
- How big is the PC-relative offset?
  - 10 bits seem sufficient in most cases



jump table

Hash Table

```
switch(x){
  case 10:
  case 20:
  case 30:
  default:
}
```


label to target addr

A[f(x)]

loaded in a register and jump to that addr indirect jump

implicit register usage

# Branch conditions

- How to specify branch conditions?
  - Condition code: test special bits set by ALU operations; creates an implicit dependence for branch (makes instruction re-ordering hard); found in x86, ARM, PowerPC, SPARC
  - GPR: comparison result put in one of the GPRs; very simple; explicit dependence; found in MIPS, Alpha
  - Compare and branch: some flavors of comparison are fused with branch instruction; one instruction per branch instead of two; complicated comparisons may affect CPI; found in VAX, PA-RISC, MIPS, Alpha
  - Turns out that a large number of comparisons are against zero beqz bnez blez bgez
  - LT and LEQ are very frequent (loop control) < <=



only call instr will do this. hardware will do as part of call instr.

# Procedure call

- What must happen on a call
  - Need to save return address somewhere; normally it is a dedicated link register or some arbitrary GPR
  - May need to set up the parameters; some architectures implicitly do this as part of call while most generate explicit code for this
  - Caller may want to save some registers so that the callee cannot destroy them; this is known as caller saving convention
  - Similarly, callee may save any register that it wants to use in later part of the procedure; this is known as callee saving convention
  - May be hard for compiler to decide what to use (interprocedural analysis is hard)
  - Most architectures offer both today with clearly specified caller saved and callee saved register sets



# Instruction encoding

- Implication on code size, memory requirement and power
  - All instructions have an opcode field to specify the operation
  - Important decision is how many bits for addressing modes
  - Number of registers, number of addressing modes, and the total number of operands in an instruction have significant impact on code size
  - Too many operands and addressing modes may necessitate separate address specifier field for each operand (what mode is used for this operand?) [VAX, x86]
  - With few addressing modes, that can be encoded in the opcode itself



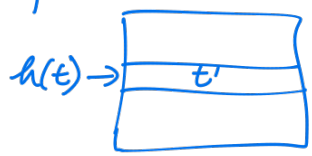
# Instruction encoding

- Fixed vs. variable length instructions
  - Fixed length instructions have a fixed number of operands (e.g. classical three-operand) and a few addressing modes (two or three) that can be encoded in opcode
  - Fixed length instructions sacrifice in terms of code size to gain in terms of complexity and performance (easy to decode)
  - Variable length instructions (e.g. 1 to 17 bytes in x86) require complicated decoders; offers a lot of addressing modes; very compact code
  - Possible to do a hybrid encoding e.g. ARM Thumb and MIPS MIPS16 offer 16 and 32-bit encoding; 16-bit encoding is used for instructions with small immediates, a subset of registers, two-operand format

Fetch compressed instr
↓
decompress (on the fly)
↓
decode
↓
branch to addr t (now at addr t something else due to compression)

Problem

Compression algo also creates a hash table where decompressed addr t is the key & it tells the new compressed instr address.




Reduced / Complex Instr Set Computer

# Instruction encoding

- IBM CodePack
  - Run length compression of PowerPC instructions on-the-fly
  - Decompress while filling instruction cache
  - For branches, uses a hash table mapping branch targets to compressed addresses
  - 10% performance loss for 35% to 40% code compaction
- In summary
  - For code size, memory and power pick variable length or hybrid or narrow encoding (last two are somewhat popular in embedded market); x86 is the only standing example of variable size ISA, but all Intel/AMD processors internally convert these to fixed size "micro-operations" for ease of implementation (more later)
  - For performance, pick fixed length encoding



n
0000...0
1111....1
log2 n+1 bits to tell how many 0's or 1's in chain
exponential saving.
(Lossless compression)

when OS brings code from hard disk to memory then compresses and brings.

# RISC vs. CISC

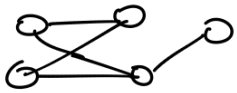
- No formal definition
- Topic of long-standing debate especially since the most successful ISA is CISC
- RISC is characterized by MIPS, RISC V-5
  - Small number of fixed length instructions (provide primitive to the compiler, not the solution)
  - Small number of addressing modes and many fast on-chip registers (make it easy for the compiler to carry out trade-off analysis among the available options)
  - Equal amount of time to execute same type of instructions (very important for pipelining and other hardware optimizations)
- The best standing example of CISC is x86
  - Today Intel processors convert x86 ISA to RISC-like micro-ops for aggressive pipelining

100 instr

MIPS → only indirect addr mode

400-500 instr

Compiler gets confused if it has many options



Register allocn → graph col. algorithm (NP hard) so we use heuristics

(variables on nodes) and edges if live interval of 2 variables intersect. Then set of colours ≡ registers

# Help the compiler guy

- The compiler is very important today
  - Your great architecture may dramatically lose in market if a good compiler cannot be designed easily
  - Offer at least 16 GPRs for integer and floating-point (graph coloring heuristics don't work well for small number of colors)
  - Addressing modes, operations, and data types should be orthogonal i.e. each mode should be applicable to all operations and all data types
  - Provide primitives, not solutions
  - Simplify trade-offs between alternatives
  - Offer instructions to bind compile-time constants i.e. avoid interpreting them at run-time
  - Less is more in the design of ISA



(MIPS has only 1 addr mode)

(more instr v/s complex instr)

if graph col not possible by compiler then spill to memory (ins, mem ops)

# Summary

- Instruction set architecture is the first step to designing a processor
  - Decide what instructions are important
  - May have to change from time to time during early phase of project; but should be finalized as soon as possible
  - Also gives the compiler team a fair idea about what to do
  - Late addition/deletion in ISA may be needed due to complexity/performance reasons



Simulator team too