Instruction Sets

"Instruction set architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine"
- IBM introducing 360 in 1964

Instruction set aspects

- operands, memory issues, operations (mostly control)

Examples

- DEC VAX, Intel IA-32, & H&P DLX

Compilers 101

Operand Storage

Why in the processor?

- faster access
- shorter address

Accumulator

+ less hardware

– high memory traffic

– likely bottleneck

Operand Storage

Stack - LIFO (60’s - 70’s)

+ code density (top of stack implicit)
– bottleneck while pipelining (why?)
• note: JAVA VM stack-based

Registers - 8 to 256 words

+ flexible: temporaries and variables
– registers must be named
– code density and “second” name space

Caches vs. Registers

Registers

+ faster (no addressing modes, no tags)
+ deterministic (no misses)
+ can replicate for more ports
+ short identifier
– must save/restore on procedure calls
– can take address of a register (distinct from memory)
– fixed size (FP, strings, structures)
– compilers must manage (an advantage?)

Registers vs. Caches

How many registers? more =>

+ hold operands longer (reducing memory traffic + run time)
– longer register specifiers (except with register windows)
– slower registers
– more state slows context switches

Operands for ALU Instructions

ALU instructions combines operands

Number of explicit operands

- two - $r_i := r_i$ op r_j
- three - $r_i := r_j$ op r_k

operands in registers or memory

- any combo - VAX - orthogonal but variable length intrs
- at least one register - IBM 360/370 - not orthogonal
- all registers - Cray, RISCs - orthogonal but loads/stores

Endian Wars

Order of bytes in words

- Big endian - MSB at address xxxx00
- Little endian MSB at address xxxx11

Big endian - IBM, Motorola, SPARC

Little endian - DEC, Intel

- Windows NT requires this (pathetic!)

Mode selectable

- becoming more common: PowerPC, MIPS

Operand Alignment

What is alignment?

- address mod size = 0
- natural boundaries

e.g., aligned word (4 bytes)

- 10 11 12 13 20
- d0 d1 d2 d3 -

e.g., unaligned word (4 bytes)

- 10 11 12 13 20
- \- d0 d1 d2 d3

Alignment

No restrictions

- simpler software
- hardware must detect misalignment
- and make 2 memory accesses
- expensive logic, slows down all references (why?)
- sometimes required for backward compatibility

Alignment

Restricted alignment

- software must guarantee alignment
- hardware only detects misalignment and traps
- trap handler does it

Middle ground

- misaligned data ok but requires multiple instructions
- compiler must still know
- still trap on misaligned access

Addressing Modes

- register: R_i — displacement $M[R_i + \#n]$
- immediate: #n — register indirect $M[R_i]$
- indexed: $M[R_i + R_j]$ — absolute: M[#n]
- memory indirect: $M[M[R_i]]$ — auto-increment: $M[R_i]$; R_i += d
- auto-decrement: $M[R_i]$; R_i -= d
- scaled: $M[R_i + \#n + R_j * d]$
- update: $M[R_i = R_i + \#n]$

modes 1-4 account for 93% of all VAX operands [Clark and Emer]

Operations

arithmetic and logical	- and, add
data transfer	- move, load
control	- branch, jump, call
system	- system call, traps
floating point	- add, mul, div, sqrt
decimal	- addd, convert
string	- move, compare

multimedia? 2D, 3D? e.g., Intel MMX and Sun VIS

Control Instructions

aspects

- 1. taken or not
- 2. where is the target
- 3. link return address
- 4. save or restore

instructions that change the PC

- (conditional) branches [1-2], (unconditional) jumps [2]
- function calls [2,3,4], function returns [2,4]
- system calls [2,3,4], system returns [2,4]

Taken or Not?

compare and branch

+ no extra compare, no state passed between instructions
– requires ALU op, restricts code scheduling opportunities

implicitly set condition codes - Z, N, V, C

+ can be set “for free”
– constraints code reordering, extra state to save/restore

explicitly set condition codes

+ can be set “for free”, decouples branch/fetch from pipeline
– extra state to save/restore

Taken or Not

condition in general-purpose register

+ no special state but uses up a register

– branch condition separate from branch logic in pipeline

some data for MIPS

- > 80% branches use immediate data, > 80% of those zero
- 50% branches use == 0 or <> 0

compromise in MIPS

- branch==0, branch<>0
- compare instructions for all other compares

Where is the Target?

arbitrary specifier

+ orthogonal

– more bits to specify, more time to decode

– branch execution and target separated in pipeline

PC relative with immediate

+ position independent, target computable in branch unit

+ short immediate sufficient - #bits: <4 (47%), <8 (94%)

– target must be known statically, can't jump far

– other techniques needed for returns, distant jumps

Where is the Target

Register

+ short specifier, can jump anywhere, dynamic target ok (ret)

– extra instruction to load register

– branch and target separated in pipeline

vectored trap - critical for OS calls

+ protection

– surprises implementation headache

Where is the Target

Common compromises

- branches - pc-rel
- jumps - pc-rel+reg
- function calls - pc-rel, reg
- returns - reg
- system calls - trap
- OS returns - reg

Link Return Address

implicit register - many recent architectures use this

+ fast, simple

– s/w save register before next call, surprise traps?

explicit register

+ may avoid saving register

– register must be specified

processor stack

+ recursion direct

– complex instructions

Save or Restore state?

what state?

- function calls: registers
- system calls: registers, flags, PC, PSW, etc

hardware need not save registers

- caller can save registers in use
- callee save registers it will use

hardware register save

- IBM STM, VAX CALLS
- faster?

Save or Restore State?

many recent architectures do no register saving

or do implicit register saving with register windows (SPARC)

Notation

generic assembly code

- sub r1, r2, r3
- means r1 := r2 - r3

data sizes

- byte 8 bits
- halfword 16 bits
- word 32 bits
- doubleword 64 btis
- quad word 128 bits

VAX

DEC 1977 VAX-11/780

upward compatible from PDP-11

32-bit words and addresses

virtual memory

16 GPRs (r15 PC r14 SP), CCs

extremely orthogonal and memory-memory

decode as byte stream - variable in length

- opcode: operation, #operands, operand types

VAX

data types

- 8, 16, 32, 64, 128
- char string - 8 bits/char
- decimal - 4 bits/digit
- numberic string - 8 bits/digit

VAX

addressing modes

- literal 6 bits
- 8, 16, 32 bit immediates
- register, register deferred
- 8, 16, 32 bit displacements
- 8, 16, 32 bit displacements deferred
- indexed (scaled)
- autoincrement, autodecrement
- autoincrement deferred

VAX

operations

- data transfer including string move
- arithmetic and logical (2 amd 3 operands)
- control (branch, jump, etc)
    - AOBLEQ
- function calls save state
- bit manipulation
- floating point - add, sub, mul, div, polyf
- system - exception, VM
- other - crc (cyclic redundancy check), insque (insert in Q)

VAX

addl3 R1,737(R2),#456

byte 1:	addl3
byte 2:	mode,R1
byte 3:	mode, R2
byte 4-5:	737
byte 6:	mode
byte 7-10:	456

VAX has too many modes and formats

Big deal with RISC is not fewer instrs

few modes/formats => fast decoding to facilitate pipelining

Intel “x86” (IA-32)

Selected history of most popular architecture

- 1974: 8080 8-bit accumulator architecture
- 1978: 8086 16-bit “kludge” architecture (ugly but early)
- Also 8088 is 8086 w/ 8-bit memory bus chosen for IBM PC
- 1985: 80386 defines 32-bit ISA but is upward compatible
- 1989: 80486, 1993: Pentium, 1995: Pentium Pro
- 1997: MMX multimedia extensions
- 200X: Superseded by IA-64 (Merced, McKinley, etc.)???

Intel “x86” (IA-32)

“Difficult to explain and impossible to love” --an x86 designer

See H&P Appendix D.8

Eight 32-bit registers (EAX, EBX, ..., but also ESP & EBP)

Also 16- and 8-bit versions (AX, AH, AL)

Most instructions have two operands, one possibly from memory

One super-duper addressing mode w/ effective address =

- base_reg + (index_reg * scaling_factor) + displacement

Many formats - see H&P fig D.8

Intel MMX

MultiMedia eXension to IA-32 [Peleg & Weiser, IEEE Micro, 8/96]

- Multimedia data values often need much less than 32 bits
- But are organized in groups (e.g., red/green/blue)
- So in 64-bit FP registers: 2x32, 4x16, or 8x8

E.g., ADDB (for byte)

```
•    17  87  100 ... 6 more
•  + 17  13  200 ... 6 more
•   --- --- ---- ... ------
•    34 100  255 (saturating or 44 with wraparound)
```

MMX takes 16-element dot product (a0*b0 + a1*b1 + ... + a15*b15)

- from 200 to 16 instructions & from 76 to 12 cycles (6x)

DLX

H&P 1990

- RISC strongly related to MIPS
- 32-bit byte addresses aligned
- load/store - only displacement addressing
- standard datatypes
- 3 fixed length formats
    - 32 32-bit GPRs (r0 = 0)
    - 16 64-bit (32 32-bit) FPRs
    - FP status register
    - no CCs

DLX

data transfer

- load/store word, load/store byte/halfword signed?
- load/store FP single/double
- moves between GPRs and FPRs

ALU

- add/subtract signed? immediate?
- multiply/divide signed?
- and,or,xor immediate?, shifts: ll, rl, ra immediate?
- sets immediate?

DLX

control

- branches == 0, <> 0
- conditional branch testing FP bit
- jump, jump register
- jump & link, jump & link register
- trap, return-from-exception

FP

- add/sub/mul/div single/double
- fp converts, fp set

DLX

	6	5	5	16
I-type:	Opcode	rs1	rd	Immediate

	6	5	5	5	11
R-type:	Opcode	rs1	rs2	rd	func

	6	26
J-type:	Opcode	Offset added to PC

I format - ALU immediate, loads/stores, branches, jump register

R format - RRR ALU ops

J format - unconditional jumps (& link?)

Compilers 101 (paper #5)

Wm. Wulf's " compilers and architecture"

Compiler goals:

- all correct programs execute correctly
- most compiled programs execute fast (optimizations)
- fast compilation
- debugging support

Compilers 101

Phases to manage complexity

Parsing --> intermediate representation
Procedure inlining
Loop Optimizations
Common Sub-Expression
Jump Optimization
Constant Propagation
Register Allocation
Strength Reduction
Pipeline Scheduling
Code Generation --> assembly code

Compilers 101

Procedure inlining	10%
local optimization	5%
register allocation	21%
global + local	14%
global+local+reg-alloc	63%
everything	81%

local: common subexpression, constant propagation

global: common subexpression, loop invariant code motion

Compiler 101

what compiler writers want:

- regularity
- orthogonality
- composability

compilers perform a giant case analysis

- too many choices make it hard

orthogonal instruction sets

- operation, addressing mode, data type

Compiler 101

one solution or all possible solutions

- 2 branch conditions - eq, lt
- or all six - eq, ne, lt, gt, le, ge
- not 3 or 4

Primitives NOT solutions

“. . . by giving too much semantic content to the instruction, the machine designer made it possible to use the instruction only in limited contexts. In many cases the complex instructions are synthesized from more primitive operations, which if the compiler had access to, could be recomposed to more closely model the feature actually needed.”

RISC vs. CISC

Clark& Bhandarkar ASPLOS paper: VAX 8700 vs MIPS R3000

combines 3 features

- architecture
- implementation
- compilers and OS

argues that

- implementation effects are second order
- compilers are similar
- RISCs are better than CISCs: fair comparison?

RISC vs. CISC

RISC factor: $\{CPI_{VAX} * Instr_{VAX}\} / \{CPI_{MIPS} * Instr_{MIPS}\}$

Benchmark	instruction ratio	CPI MIPS	CPI VAX	CPI ratio	RISC factor
li	1.6	1.1	6.5	6.0	3.7
eqntott	1.1	1.3	4.4	3.5	3.3
fpppp	2.9	1.5	15.2	10.5	2.7
tomcatv	2.9	2.1	17.5	8.2	2.9

RISC vs. CISC

compensating factors

- increase VAX CPI but decrease VAX instruction count
- increase MIPS instruction count
- e.g. 1: loads/stores vs. operand specifiers
- e.g. 2: necessary complex intructions: loop branches

factors favoring VAX

- big immediate values
- not-taken branches incur no delay

RISC vs. CISC

factors favoring MIPS

- operand specifier decoding
- number of registers
- separate floating point unit
- simple branches/jumps (lower latency)
- no complex instructions
- instruction scheduling
- translation buffer
- branch displacement size