Chapter 3: Instruction Sets

(Instruction Set) Architecture is the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

--IBM, introducing 360 in 1964
(eliminating 7 instrns sets)

Instruction Set Aspects

- Operands
- Memory issues
- Operations (mostly control)

Compilers 101 & [Wulf81]

Examples

- VAX
- 8086
- DLX
- Some numbers

Alpha and PowerPC (no slides)

Operand Storage in CPU

Why in CPU?

- Faster access
- Shorter, sometimes implicit, address

Accumulator -- one implicit register (< 1960)

- + Minimum hardware resources
- - High memory traffic

Stack -- LIFO storage (1960s-1970s)

- + Excellent code density (TOS implicit)
- - Stack can become bottleneck, especially with pipelining

Registers -- 8 to 256 words (1960s-???)

- + Flexible: temporaries and variables
- - Register must be named

Caches vs. Registers [Ditz82]

Register advantages

- Faster (no addressing mode, no tags)
- Deterministic (no misses)
- Can duplicate for two ports
- Short identifier (3-8 bits)

Register disadvantages

- Must save/restore on procedure calls
- Can't take the address of a register
- Fixed size (FP, strings, structures)
- Compiler must control (an advantage?)

How many registers? More means

- + Hold operands longer (reducing memory traffic & potentially execution time)
- - Longer register specifiers (except with register windows)
- - Slower registers
- - More state slows context switches

Operands for ALU Instructions

ALU instructions combine operands (e.g. ADD)

Number of explicit operands

- Two -- destination equals one source
- Three -- orthogonal

Operands in registers or memory

- Any combination -- VAX
  - (orthogonal, but variable instrn formats)
- At least one register -- much of 360
  - (not orthogonal)
- All registers -- CRAY, DLX, RISCs
  - (orthogonal, but need loads/stores)

Endian Wars

Order of bytes in words

Big Endian -- MSB at address xxxxx00

Little Endian -- MSB at address xxxxx00

Big Endian (IBM, Motorola)

Word Address	MSB			LSB
0	0	1	2	3
4	4	5	6	7
	0	Bit Position		31

Little Endian (DEC, Intel)

Word Address	MSB			LSB
0	3	2	1	0
4	7	6	5	4
	0	Bit Position		31

Operand Alignment

What is alignment?

Address mod Size = 0

e.g. Aligned word (4 bytes)

	10	11	12	13	14	
	DO	D1	D2	D3		

e.g. Unaligned word (4 bytes)

	10	11	12	13	14	
		DO	D1	D2	D3	

Restricted Alignment

No Restrictions

Software is simple

Hardware must detect misalignment and (typically) make 2 memory access.

Expensive logic

Usually slows down all references

Restricted Alignment

Software must guarantee alignment

Hardware only detects misalignment and traps

Middle ground (VAX 8800)

Misaligned data OK, but slow

Traps on misaligned access, 10 cycle penalty

Addressing Modes

1)	Register	Ri
2)	Displacement	M[Ri + #n]
3)	Immediate	#n
4)	Register Indirect	M[Ri]
5)	Indexed	M[Ri + Rj]
6)	Absolute	M[#n]
7)	Memory Indirect	M[M[Ri]]
8)	Auto-Increment	M[Ri]; Ri = Ri + d
9)	Auto-Decrement	Ri = Ri + d; M[Ri]
10)	Scaled	M[Ri + #n + Rj * d]
new!)	Update	M[Ri = Ri + #n]

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

Operations

Arithmetic & Logical	AND, ADD
Data Transfer	MOVE, LOAD
Control	BRANCH, JUMP, CALL
System	OS CALL, VM
Floating Point (opt)	+,-,*,/,sqrt
Decimal (opt)	ADDD, CONVERT
String (opt)	MOVE, COMPARE

Why implement optional instructions?

Control Instructions

Aspects

(1) Taken or not taken?

(2) Where is the target?

(3) Link return address?

(4) Save or restore state?

Instructions that ‘‘change the PC’’

(Conditional) branches (1, 2)

(Unconditional) jumps (2)

Procedure calls (2, 3, 4)

Procedure returns (2, 4)

O.S. calls (2, 3, 4)

O.S. returns (2, 4)

(1) Taken or not taken?

Compare and branch instruction

+ No extra compare instruction
+ No state passed between instructions
- Requires ALU operation (so does (2))

Condition codes (Z,N,V,C)

+ Can be set for free
- Constrains code reordering
- Extra state to save and implement

Condition in general-purpose register

+ No special state to save and implement
- Uses up a register

(1) Taken or not taken?, cont.

Some data for MIPS

>80% of compares for branches use immediates

>80% of these immediates are zero

50% compares for branches are =0 or ≠0

Compromise used in MIPS

Have branch-if-=0 and branch-if-≠0

Have compare instructions ($r1 = r2$, $r1 \neq r2$, $r1 < r2$, $r1 \leq r2$, etc.) set a register.

With pipelining, can we predict whether taken?

Statically?
Dynamically?

(2) Where is the target?

Could use Arbitrary Specifier

+ Orthogonal and powerful
- More bits to specify
- More time to decode

PC-relative with immediate

+ Position independence (helps linking)

+ Short immediate sufficient. MIPS word immediates:
  None equal zero
  ≤ 4 bits: 47%
  ≤ 8 bits: 94%
  ≤ 12 bits: 100%

- Target must be known statically (to link)

- Can't jump arbitrarily far

- Other techniques are required for returns and distance jumps

(2) Where is the target?, cont.

Register

+ Short specification
+ Can jump anywhere
+ Dynamic target okay (returns)
- Extra instruction to load register

(Vectored) Trap

Critical for O.S. calls
+ Protection, protection, protection
- Implementation headache

Common compromise

(Conditional) branches (pc-rel)
(Unconditional) jumps (pc-rel, reg)
Procedure calls (pc-rel, reg)
Procedure returns (reg)
O.S. calls (trap)
O.S. returns (reg)

(3) Link return address?

Required for procedure calls and O.S. calls

Implicit register

+ Fast, simple
- SW must save register before next call
- Surprise traps or interrupts?

Explicit register

- No important advantages over above
- Register must be specified

Processor Stack

+ Recursion supported directly
- Complex instruction

Many recent architectures use implicit register

(4) Save or restore state?

What state?

Procedure calls: registers
O.S. calls: registers and PSW (incl. CCs)

Hardware need not save registers

Caller can save registers in use
Callee can save registers it will use

Hardware register save

Which (IBM STM, VAX CALLS)?
Is the above faster?
Register windows (Chapter 8)

Many recent architectures do no register saving or do implicit saving with register windows

(I have ignored parameter passing)

Compilers 101

Compiler Goals

- All correct programs compiled correctly
- Most compiled programs execute fast
- Fast compilation
- Small code size
- Debugging support, etc.

Multiple Source Compilers

- Same compiler can compile different languages

Multiple Target Compilers

- Same compiler can generate code for different target machines

Compilers 101, continued

Phases used to manage complexity

Front-end (converts source to intermediate form)

High-level optimizer

- Procedure inlining, loop transformations

Global optimizer

- Common subexpression elimination, constant propagation, induction variable elimination, and register **allocation**

Code-generator (and assembler)

- Strength reduction, pipeline scheduling

Compilers 101, cont.

Where are variables?

- Stack -- activation records
- Global data area -- static global (arrays)
- Heap -- dynamic objects (via pointers)

Register allocation very important

- Can't allocate if address is needed
- Use graph coloring (NP-Complete)
- Register lives adjacent if overlapping
- Want many general-purpose registers (colors $\geq$ 16) or else heuristics fail

Compilers 101, cont.

Optimizations	Percent Faster
Procedure Integration	10%
Local Optimization	5%
Local Opt and Reg Allocation	26%
Global and Local Opt	14%
Global and Local Opt and Reg Alloc	63%
Everything	81%

Local Optimizations

- Common Subexpression Elimination
- Constant Propagation

Global Optimizations

- Common Subexpression Elimination
- Code Motion (Loop Invariants)

Compilers 101 [Wulf81]

What Compiler Writers want:

- Regularity
- Orthogonality
- Composibility

Compilers perform a GIANT case analysis

- Too many choices make it hard

Orthogonal Instruction Sets

- Operation
- Addressing Mode
- Data Type

Compilers 101, [Wulf81]

One solution, or All possible solutions

- 2 Branch Conditions (EQ,LT)
- Or all 6 (EQ,LT,GT,NE,LE,GE)
- Not 3 or 4

Primitives, NOT Solutions

> Semantic Clash: ''...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 of these cases, the high-level instructions are synthesized from more primitive operations which, if the compiler writer could access them, could be recomposed to more closely model the features actually needed.''

Notation

Generic Assembly Code

```
SUB   R1,R2,R3
```

Means

$R1 \leftarrow R2 - R3$

Data Sizes

Byte	8 bits
Halfword	16 bits
Word	32 bits
Double Word	64 bits
Quad Word	128 bits

VAX

Introduced by DEC in 1977: VAX-11/780

Upward compatible from PDP-11

32-bit ''word'' and addresses

Virtual memory is first-class

16 GPRs (r15 is PC, r14 is SP), CCs

Extremely orthogonal, memory-memory

Decode as byte stream

- Opcode: operation, number of operands & operand type
- Variable-length address specifiers

VAX, cont.

Data types

- 8-, 16-, 32, 64, 128-bit integers
- F (32 bits), D (64), G (64), H (128) FP
- Character string (8-bits/char)
- Decimal (4-bits/digit)
- Numeric string (8-bits/digit)

Addressing modes include

- Literal (6 bits)
- 8-, 16-, 32-bit immediates
- Register, register deferred
- 8-, 16-, 32-bit displacements
- 8-, 16-, 32-bit displacement deferred
- Indexed (scaled)
- Autoincrement
- Autodecrement
- Autoincrement deferred

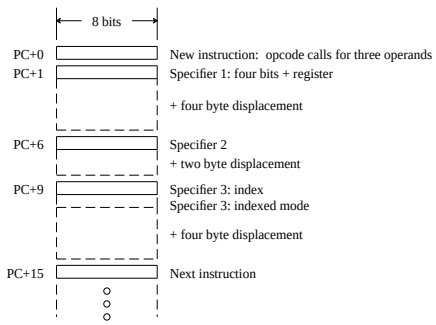
VAX, cont.

Operations

- Data Transfers (including string move)
- Arithmetic and Logical (2 and 3 operands)
- Control (Branch, Jump, etc)
  - AOBLEQ (Add One and Branch if Less than or EQual)
- Procedure (CALLS saves state)
- Bit Manipulation
- Floating Point (Add/Sub/Mult/Divide)
  - POLYF -- Polynomial Evaluation
- System (Exception, VM)
- Other
  - CRC -- Cyclic redundancy check
  - INSQUE -- Insert entry in queue

VAX, cont.

VAX byte stream:



VAX has too many modes & formats

- Serial semantics limit parallel interpretation
- The big deal with RISC is not REDUCED numbers of instructions; it's few modes & formats to facilitate pipelining

8086

Introduced by Intel in 1978

- 8088 (8-bit external bus) chosen for IBM PC in 1980
- Most popular general-purpose machine

16-bit microprocessor

Upward from 8080 (accumulator-based)

Terrible architecture

- Special-purpose registers: 4 arithmetic, 4 address, 4 segment, 2 control
- Addresses: (16b segment)<<4 + 16b offset (i.e. 64K 16KB-aligned 64KB segments)
- Many formats (see H&P Figure 4.13)

80x86s ('81, '82, '85, '89) -- compatibility mode

DLX

Introduced by H&P in 1990

A RISC architecture strongly related to MIPS

32-bit byte addresses (aligned)

Load/Store -- only displacement addressing

Standard datatypes

Registers

- 32 32-bit GPRs (r0=0)
- 16 64-bit (32 32-bit) FPRs
- FP Status Register
- no CCs

Three fixed-length formats

DLX, cont.

Instruction Set -- Part 1 of 2

Data transfer

- Load/store word
- Load/store byte/halfword signed?
- Load/store FP single/double
- Moves between GPRs & FPRs

ALU

- Add/subtract signed? immediate?
- Multiply/divide signed?
- AND, OR, XOR immediate?
- Load high (not real load)
- Shifts; LL,RL,RA; immediate?
- Sets immediate?

DLX, cont.

Instruction Set -- Part 2 of 2

Control (not delayed)

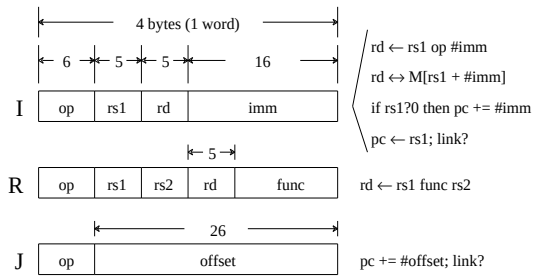
- Conditional branch =0 & ≠0
- Conditional branch testing FP bit
- Jump, jump register
- Jump & link, jump & link register
- Trap, return-from-exception

FP

- Add/subtract/multiply/divide single/double
- FP converts
- FP sets (sets FP bit)

DLX, cont.

Three fixed-length formats:



I-Format

- ALU ops with immediates
- loads & stores
- conditional branches
- jump register (& link to r31?)

R-Format: RRR ALU ops

J-Format: unconditional jumps (& link?)

DLX Numbers

Type	gcc	espresso	spice	nasa7
int load/store	43%	29%	23%	1%
int arith	26%	30%	33%	22%
control	17%	13%	11%	4%
logical	10%	23%	5%	1%
fp load/store	0%	0%	8%	33%
fp arith	0%	0%	5%	39%
misc	4%	4%	5%	0%

Fraction time / fraction instrns
(DECstation 3100 w/ MIPS R2000)

- FP: 2/1
- Data transfer: 1.5/1
- ALU: 0.8/1 to 1/1
- Control: 1.3/1

Note: ratios are near unity

Bhandarkar and Clark: RISC vs. CISC

Compares the VAX 8700 vs MIPS M/2000 (R3000 chip)

Combines 3 factors

- Architecture
- Implementation
- Compilers and OS

Argues that

- Implementation effects are second order
- Compilers are *similar*
- RISCs are better than CISCs

Is this a fair comparison of RISCs vs CISCs???

Bhandarkar and Clark, cont.

RISC Factor

$$RISC\ factor = \frac{CPI_{vax}\ Instr_{vax}}{CPI_{mips}\ Instr_{mips}}$$

Benchmark	Instr Ratio	MIPS	CPI VAX	ratio	RISC factor
spice2g6	2.5	1.8	8.0	4.4	1.8
matrix300	2.4	3.0	13.8	4.5	1.9
nasa7	2.1	3.0	15.0	5.0	2.4
fpppp	2.9	1.5	15.2	10.5	2.7
tomcatv	2.9	2.1	17.5	8.2	2.9
dudoc	2.7	1.7	13.2	7.9	3.0
espresso	1.7	1.1	5.4	5.1	3.0
eqntott	1.1	1.3	4.4	3.5	3.3
li	1.6	1.1	6.5	6.0	3.7
geo mean	2.2	1.7	9.9	5.8	2.7

Bhandarkar and Clark, cont.

Compensating Factors

- Increase VAX CPI but decrease VAX instruction count
- Increase MIPS instruction count
- Example 1: Loads and stores vs. operand specifiers
- Example 2: Necessary complex operations, e.g. loop branches

Factors favoring VAX

- Big immediate values
- Not-taken branches incur no delay

Bhandarkar and Clark, cont.

Factors favoring MIPS

- Operand specifier decoding
- Number of registers
- Separate floating point unit
- Simple jumps and branches (lower latency)
- Fancy VAX instructions: Unnecessary functionality
- Instruction scheduling
- Translation buffer
- Branch displacement size

DEC Alpha

Introduced by DEC in 1992

- Ability to emulate VAX instructions important
- Strongly influenced by Cray-1

64-bit architecture

Load/Store -- only displacement addressing

Standard datatypes

- No byte loads/stores

Registers

- 32 64-bit GPRs (r31=0)
- 32 64-bit FPRs

VAX and IEEE floating point

DEC Alpha, contd.

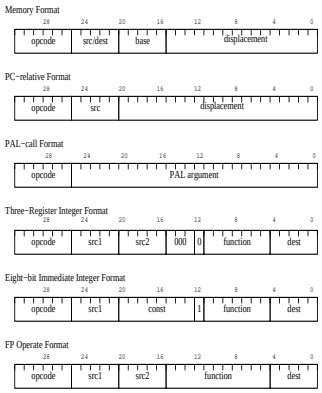
Four fixed-length instruction formats

- Sub-formats for computation instructions
- 32-bit instructions
  Designed with multiple-issue in mind
  No delayed branches

Precise exceptions not automatic

PAL code

DEC Alpha Instruction Formats



DEC Alpha Instruction Set

Operate Instructions

- Integer arithmetic
- Logical (AND, OR, conditional MOV)
- Byte-manipulation
- Floating-point arithmetic
- Miscellaneous (memory prefetching, trap and memory barriers)

Load/Store Instructions

- Load/Store Quadword (64 bits)
- Load-Linked/Store Conditional (for MP synchronization)

DEC Alpha Instruction Set, contd.

Control/Branching Instructions

- Branch on condition (8 conditions) in integer register
- Branch on condition (6 conditions) in FP register
- Unconditional branches
- Calculated jumps

Branching Hints

- Different hint/rule for type of branch

Supervision Instructions

- PAL code for needed task

Conditional Move Instructions

C-code fragment

```
if (a > b) {
    flag = a;
}
```

Without Cond. Moves

```
    subu    cond, a, b
    br_gt   L1
    mov     flag, a
L1:
```

With Conditional Moves

```
    subu     cond, a, b
    cmov_eq  flag, a, cond
```