Instruction Set (ISA) Design and Addressing Modes

To be covered:

- ISA Design Overview (2.1)
- Architecturally visible storage.
- ISA classification based on register organization. (2.2)
- Addressing modes. (2.3)
- Impact on ISA.

ISA Design Overview

ISA Design Choices (In reverse order.)

- Instruction Operations (`ADD, SUB,` etc.)
- Instruction Forms and Coding (number of operands, etc.)
- Specifying Operands (we'll get into that)

Instruction Operations

- Arithmetic and Logical Instructions
  You know, `ADD`, `MUL`, `XOR`.
- Control Transfer
  Collective term for branches, jumps, and subroutine calls.
- Data Movement (Load/Store, Register Movement)
  Includes, register ↔ register, register ↔ memory, memory ↔ memory.
- Process Management (Operating System, etc.)
  For use by OS to control interaction of programs.

Specifying Operands: Architecturally Visible Storage

Consider: `ADD` ⟨**sum**⟩ = ⟨**op1**⟩ + ⟨**op2**⟩

Operands ⟨**op1**⟩ and ⟨**op2**⟩ can refer to:

- A Constant
- Something Written Earlier

Since "Something Written Earlier" is part of instruction . . .

. . . the ISA must define names for that storage.

Since storage defined by ISA it's called *architecturally visible storage*.

Common types of architecturally visible storage:

- Registers
  Sometimes there are multiple sets.
- Memory
  Sometimes there are multiple address spaces.

Other types are less common.

What ISA Defines for Architecturally Visible Storage

- Names.
  For registers, *r1*, *f30*, *g6*. For memory, 53023.
- Result of writing and reading storage.
  Not obvious with multiple readers and writers.

Registers and Memory

Registers (Internal Storage)

Store what is actively being worked on.
E.g. Math expression parts, array indices.

Implemented using highest speed memory.

Given short names.
E.g. `r1, g1, AL`.

Small number of registers provided.
E.g. 32, 64.

Goal: fastest access.

Memory

Stores code and data.

Simple to programmer . . . despite complex implementation.

Large number of locations, $2^{32} = 4294967296$ and
$2^{64} = 18446744073709551616$ are common.

Named using integers called *addresses*. . .
. . .and some address space identifier.

Goal: large size.

Specifying Operands: ISA Classification

Consider: ADD ⟨**sum**⟩ = ⟨**op1**⟩ + ⟨**op2**⟩

What ⟨**op1**⟩ and ⟨**op2**⟩ usually refer to:

- Register contents.
- Memory contents.
- Part of instruction.
- A constant.

ISAs classified on allowable register and memory operands.

ISA Classification

Classified by allowed register and memory operands.

Listed below by distinguishing features ...
... and *intended* (contemporary) benefits.

Next slide: current relevance.

Load/Store, General-Purpose Registers (GPR)

ALU instructions refer only to registers. (Not memory.)

Memory ↔ register movement uses *Load* and *Store* instructions.

Number of special-purpose registers minimized.

⇒Keep memory and ALU operations separate.

⇒Avoid special-purpose instructions.

Memory/Memory, General-Purpose Registers

Both load/store instructions and ALU instructions ...

... can refer to both registers *and* memory.

Few special-purpose registers.

⇒Use registers only when needed.

Accumulator

Typical ALU instruction uses...
...a special *accumulator* register...
...and a general purpose register.

Only the general purpose register need be specified.

⇒Keep instructions small.

Stack

Instructions to push and pop data from stack.

ALU instructions refer to stack.

No registers in usual sense.

⇒Natural way to evaluate expressions.

Tradeoffs of ISA Types

Load/Store, General-Purpose Registers (GPR)

Used in most ISAs developed in past decade.

+ Programmer- and compiler-friendly.
  (Since most registers can be used for any purpose.)
+ Allow single-size instruction coding.
  (Since multiple memory addresses not needed in ALU instr.)

Memory/Memory, General-Purpose Registers

+ Lots of addressing options for programmers.
– Lots of addressing options forces slower implementation.

Accumulator

– Extra instructions needed to move data.

Stack

+ Programs small.
– Implementations slow.
– Hard to code certain expressions.

Common Addressing Modes

Addressing modes used by many ISAs.

Register

Data in register.

```
Move R4, R3 !  R4 = R3 (This is a comment.)
```

Immediate

Data in instruction.

```
Move R4, #3 !  R4 = 3
```

Register Deferred or *Register Indirect*

Data address in register.

```
Load R4, (R1) !  R4 = MEM[R1]
```

Displacement

Data address is register plus constant.

```
Load R4, 100(R1) !  R4 = MEM[ R1 + 100 ]
```

Indexed

Data address is sum of two registers.

```
Load R4, (R1+R2) !  R4 = MEM[ R1 + R2 ]
```

Direct

Data address is a constant.

```
Load R1, (1024) !  R1 = MEM[ 1024 ]
```


Memory Indirect

Address of data's address is in register.

```
Load R1,@(R3) !  R1 = MEM[ MEM[ R3 ] ].
```

Autoincrement

Perform register indirect access, then add constant to register.

```
Load R1,(R2)+ !  R1 = MEM[ R2 ]; R2 = R2 + 1
```

Autodecrement

Subtract constant from register then perform register indirect access.

```
Load R1,-(R2) | R2 = R2 - 1; R1 = MEM[ R2 ];
```

Scaled

Data address is constant1 + reg1 + reg2 * constant2.

```
Load R1,100(R2)[R3] !  R1 = MEM[ 100 + R2 + R3 × d ]
```

There's no limit to how many addressing modes one could think of.

Memory Addressing Choices in ISA Design

Which addressing modes?

Affects cost and may limit future performance.

Which instructions get which addressing modes?

Affects cost and may limit future performance.

Maximum displacement size?

Limited by instruction size.

Maximum immediate size?

Limited by instruction size.

Usage of Addressing Modes

Do we really need all those addressing modes?

Memory Addressing Usage in VAX Code.

VAX uses all of addressing modes described earlier.

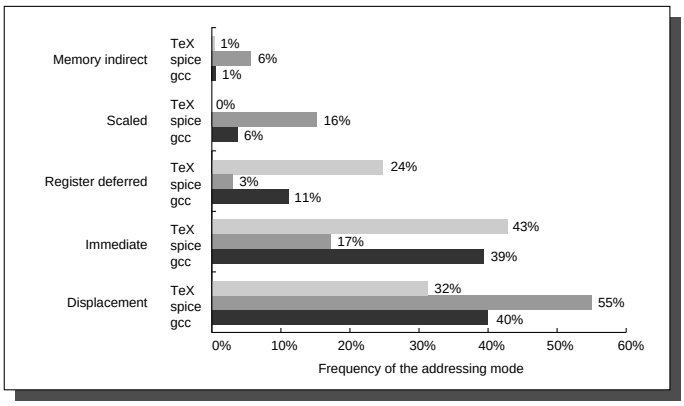

FIGURE 2.6 Summary of use of memory addressing modes (including immediates).

Modes used less than 1% of time omitted.

Large differences between programs.

Since a few modes account for most accesses . . .
. . . others could be omitted with little impact on performance . . .
. . . saving silicon area (but programs would have to be rewritten).

Displacement Sizes

What should the maximum displacement size be?

Too large: difficult to code instruction.

Too small: won't be very useful.

Displacement Size in SPECint92 and SPECfp92 Programs on MIPS.

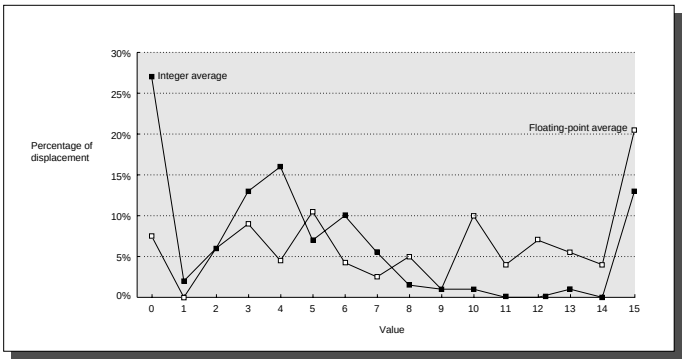

FIGURE 2.7 Displacement values are widely distributed.

Wide range of displacements used.

Immediate Sizes

What should the maximum immediate size be?

Too large: difficult to code instruction.

Too small: won't be very useful.

Immediate Sizes in VAX Code

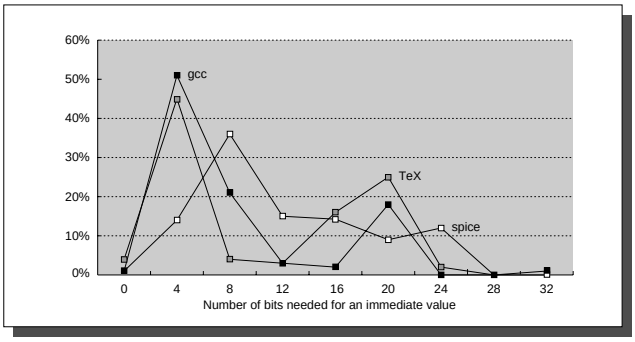

FIGURE 2.9 The distribution of immediate values is shown.

Smaller values used more frequently.

Internal Storage Variations

Operands per Instruction

Three typically used.

Two sometimes used.

Factors:

Instruction coding (bits to specify operands).

Addresses per ALU Instruction

Zero typically used (load/store).

One, two, even three sometimes.

Factors

Instruction coding.
(Addresses take up lots of space.)

Benefit over multiple instructions.

Memory Addressing

Address Interpretation

Sequence of memory locations (usually bytes) starting at address.

Size of sequence depends upon instruction.

E.g., `LW`, load word instruction might read four bytes.

E.g., `LB`, load byte instruction might read one byte.

Alignment

Addresses subject to *alignment* restrictions...
...when used in certain instructions.

E.g., a *word-aligned* address must be divisible by 4 (usual word size).

Past Choices

Vax (Digital Equipment Corporation)

Older ISA, variable instruction size.

Provides many addressing modes usable by many instructions.

Instructions use 32-bit immediate.

+ (ISA) Compact code.

– (Imp) Future implementations limited.
So limited that DEC developed a new ISA, Alpha.

– (Imp) Expensive-to-implement modes go unused.

SPARC V8 (Sun Microsystems)

Newer ISA, fixed 32-bit instruction size.

Load/Store instruction modes: indexed and 13-bit displacement.

ALU instruction modes: register and 13-bit immediate.

`sethi` instruction modes: 22-bit immediate.
(Used for moving large immediate into registers.)

+ (Imp) Uniform instruction sizes.

+ (Imp) Useful modes included.