Instruction Set (ISA) Design and Addressing Modes

Material from sections 2.1, 2.2, and 2.3.

Outline

ISA Design Choices
It’s more than just picking instructions.

ISA Design Choice Details
Screw up, and you’ll be cursed for decades.

ISA Design Decisions

I. Organization
A. Data types (supported by ISA).
B. Memory and register organization.
C. Addressing modes.

II. Instruction Choices
A. Data movement instructions.
B. Arithmetic and logical instructions.
C. Control transfer instructions (CTIs). (Branch, jump, call, return.)
D. Process and processor management instructions.

III. Instruction Coding

ISA Design Choice Details

Outline
Data Types
Memory and Register Organization
ISA Classification
Addressing Modes
Displacement and Immediate Sizes

Data Types

To include a new data type:
Define operations.
Determine its sizes.
Add new instructions to operate on it.
Integer Data Types

Needed For

- Address computation.
- Counting (loop iterations, etc).
- Arithmetic, of course.

Data Types for Simple 32-bit Machine

<table>
<thead>
<tr>
<th>Type</th>
<th>Special Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit signed integer</td>
<td>addu</td>
</tr>
<tr>
<td>32-bit unsigned integer</td>
<td>lh, load half-word.</td>
</tr>
<tr>
<td>16-bit signed integer</td>
<td>lh, load half-word signed</td>
</tr>
<tr>
<td>16-bit unsigned integer</td>
<td>lh, load byte.</td>
</tr>
<tr>
<td>8-bit signed integer</td>
<td>lh, load unsigned byte.</td>
</tr>
<tr>
<td>32-bit float</td>
<td>addf, add 32-bit floating-point.</td>
</tr>
<tr>
<td>64-bit float (double)</td>
<td>adddd, add 64-bit floating-point.</td>
</tr>
</tbody>
</table>

Signed integer types operated on by integer arithmetic instructions.

Unsigned integers operated on by logical and unsigned integer arithmetic instructions.

The basic 32-bit load instruction not appropriate for smaller types.

The \( lh, lh, lb, \) and \( lhu, lbu \) instructions ... 
... place data in low portion of 32-bit registers ... 
... and place zeros or a sign bit high portion.

Data Type Tradeoffs

Is cost worth the benefit?

Benefit is improved performance, but who benefits?:

Everyone (performance better on whole SPEC suite) ... 
... if not then some class of programs (e.g., floating point, graphics) ... 
... if not then my demo program :-(.

Chip area, a limited resource, ...
... might be given to data type (or other feature) with most benefit.

Data Type Tradeoff Examples

Start with integer-only ISA.

Example of a good new data type: floating-point.

FP hardware many times faster than software.

Floating-point arithmetic used frequently in many programs.
Example of a bad new data type: time.

Detail of time data type:

Size 64 bits. (The number of seconds since 1970 UTC, avoid Y2.038k [s4G?] problem.).

Some Instructions:

\begin{verbatim}
  t.add.day (sumtime) (time) (days) ... 
  ... All operands are registers. ... 
  ... Add (days) days (an integer) to (time) (a time), store result in (sumtime).

  t.to.dom (dom) (time) ... 
  ... All operands are registers. ... 
  ... Store the day of month (integer) for time (time) in register (dom).

  t.diff (diff) (time1) (time2) ... 
  ... All operands are registers. ... 
  ... Store the difference between (time1) and (time2) in (diff).
\end{verbatim}

Problems with time data type.

Instructions would not be used often enough.
Possibly not much faster.
Complex control, about the same as transcendental functions (sin, etc.).

Therefore chip area and unused opcodes should be used for other new data types.

Overview of Popular Data Types

Common Data Type Sizes

Size names are particular to an ISA.
Don’t assume a word size is four (or two or eight) bytes.

Type usually specified with a size.

\begin{itemize}
  \item \textit{Byte}, char, octet. 1 byte (8 bits here).
  \item \textit{Half word}. 2 bytes.
  \item \textit{Word}. 4 bytes.
  \item \textit{Doubleword}. 8 bytes.
  \item \textit{Quadword}. 16 bytes.
\end{itemize}

Common Types and Sizes

- \textit{Unsigned integer} and \textit{integer}. Byte, half word, word, doubleword.
  Used for address computation and throughout programs.

  Integer size (along with address space) defines ISA size: 32-bit, 64-bit, etc.

  Integers are sign-extended when moved into a larger register ... ... while unsigned integers are not.

- \textit{Floating-point}. Word, doubleword, quadword.
  Most newer machines use the IEEE 754 format.
• **BCD.** Word, etc.
  Each word holds several BCD digits of a *single* fixed-point number.
  E.g., word holds a 8-digit BCD integer.
  Decimal fractions such as .03 exactly represented.
  Used for financial computations, typically in Cobol programs.
  Used primarily in older architectures.

• **Packed integer, packed fixed-point.** Word, double word.
  Holds *several* small integer or fixed-point values.
  Operated on by *saturating* arithmetic instructions.
  Used by *packed-operand* instructions which operate on each small value in parallel.
  Used in newer ISA versions. E.g., Sun VIS, Intel MMX, HP PA MAX.

### Integer, BCD, and Packed Integer Example

Consider integers 1239₁₀ and 5678₁₀.

As half-word-size (short) integers: ...

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
\text{Sign} & \text{Short Int.} \\
\hline
31 & 16 & 15 & 0 \\
\end{array}
\]

\[
\ldots \quad 1239₁₀ = 0x04d7 \\ x 10 \\
\ldots \quad 5678₁₀ = 0x162e \\
\]

As BCD integers:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
\text{MSD} & \text{LSD} \\
\hline
31 & 28 & 27 & 24 & 23 & 20 & 19 & 16 & 15 & 0 \\
\end{array}
\]

\[
\ldots \quad 1239₁₀ = 0x1239 \\
\ldots \quad 5678₁₀ = 0x5678 \\
\]

Addition of packed integers is *saturating* ...

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
\text{Sign} & \text{Short Int.} \\
\hline
31 & 28 & 27 & 24 & 23 & 20 & 19 & 16 & 15 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
\text{MSD} & \text{LSD} \\
\hline
31 & 28 & 27 & 24 & 23 & 20 & 19 & 16 & 15 & 0 \\
\end{array}
\]

Addition: Integer      | BCD            | Packed Int.
---------------------|----------------|---------------------
0x04d7               | 0x1239         | 0x1239             \\
+ 0x162e             | 0x5678         | 0x5678             \\
0x1b05               | 0x6917         | 0x68af             \\
= 6917₁₀            | 6917₁₀         | (6, 8, 10, 15)     \\
0x1b05 = 0x68af      |                |                    \\
0x6917 =              |                |                    \\
0x68af =              |                |                    \\

For example, 12 + 8 = 15, assuming 15 is the maximum value.
Data Type Usage

Data below for SPEC92 programs on VAX.

![Data Type Usage Diagram]

FIGURE 2.16 Distribution of data accesses by size for the benchmark programs.

Size Tradeoffs

Integer: size of fastest integer (usually) equals address size.

E.g., word on a 32-bit machine, doubleword on a 64-bit machine.

On most machines a smaller integer saves space, but not time.

Floating-point: doubleword usually best choice.

Word may be faster, but can be slower . . .

. . . when double result must be rounded to word size.

Data Type Coding

How data type coded:

In opcode. (Used in many ISAs.)

Integer multiply instruction, floating-point add.

In instruction’s type field. (Used in many ISAs.)

Tagged, type specified in data. (Used in a few ISAs.)

Suppose data type were word-sized, . . .

. . . 30 bits might hold the number . . .

. . . 2 bits would indicate what type the data was . . .

. . . such as integer, unsigned integer, float, or string.

Memory and Register Organization

Consider: \( \text{ADD} (\text{sum}) = (\text{op1}) + (\text{op2}) \)

Operands \((\text{op1})\) and \((\text{op2})\) can refer to:

- A Constant (Immediate)
- 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.
  For systems covered in this class result is obvious (value read is last value written).
  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.

- Many locations, \( 2^{32} = 4294967296 \) and \( 2^{64} = 18446744073709551616 \) \(^1\) are common.

- \( 2^{28} = 3402823669209834656 \) is a long way off or may never be used.

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

- Goal: large size.

- Rule of thumb: address space needed grows by one bit per year.

- Very difficult to change ISA's address space size ... so chosen to be much larger than contemporary needs.

1 Eighteen quintillion, four hundred forty-six quadrillion, seven hundred forty-four trillion, seventy-three billion, seven hundred nine million, five hundred fifty-one thousand, six hundred sixteen.

2 Three hundred forty undecillion, two hundred eighty two decillion, nine hundred twenty octillion, nine hundred thirty eight septillion, four hundred sixty three sextillion, four hundred sixty three quintillion, three hundred seventy four quadrillion, six hundred thirty trillion, four hundred thirty one billion, seven hundred sixty eight million, two hundred eleven thousand, four hundred sixty six.
Alignment

Addresses may be subject to alignment restrictions...
...when used in certain instructions.

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

Example.

! In an unaligned ISA both instructions can execute.
! In an aligned ISA at most one could execute, the other would
! cause an error (memory access violation exception).

lw  r1, 0(r2)  ! Load r1 with data at address r2.
lw  r3, 1(r2)  ! Load r3 with data at address r2 + 1.

Common Addressing Modes

Addressing modes used by many ISAs.

Register

Data in register.

Move  r4, r3  ! r4 = r3  Data in r3.
add  r4, r2, r3  ! r5 = r2 + r3  Data in r2 and r3.

Useful when data recently produced and is still in register.

All ISAs with registers have register addressing.

Immediate Addressing:
Data in instruction.

Move  r4, #3  ! r4 = 3.  Data, 3, in instruction.
add  r4, r2, #3  ! r4 = r2 + 3.  Data, 3, in instruction.

All ISAs have some form of immediate addressing.

ISA design parameter: immediate size (maximum immediate value).

Common Memory Addressing Modes

Memory Addressing Modes

With memory addressing modes data is in memory.

Modes specify an effective address, the memory location at which data located.

There are many ways to specify a memory address.

Direct Addressing:
Effective address is a constant in instruction.

load r1, (1024)  ! r1 = MEM[ 1024 ]  Data at 1024.
add  r4, r2, (1024)  ! r4 = r2 + MEM[ 1024 ]

The add instruction could not be in load/store ISA.

ISA may need large instructions to accommodate the address.

Included in ISAs with variable instruction sizes.
Register Deferred Addressing: or Register Indirect Addressing
Effective address in register.

\[
\begin{align*}
\text{Load} & \text{ r4, (r1) } \quad \text{r4} = \text{MEM}[\text{r1}] \\
\text{add} & \text{ r4, r2, (r1) } \quad \text{r4} = \text{r2} + \text{MEM}[\text{r1}] \\
\end{align*}
\]

Note: the add instruction could not be in load/store ISA.

Included in most ISAs.

Displacement:
Effective address is register plus constant.

\[
\text{Load r4, 100(r1) } \quad \text{r4} = \text{MEM}[\text{r1} + 100] \\
\]

Useful for accessing elements of a structure:

\[
\begin{align*}
! \text{In c: struct } \{ \text{int } i; \text{ short int } j; \text{ unsigned char } c; \} \text{ str;} \\
! \text{r1 = } \&\text{str}; \\
\text{lw } &\text{ r2, 0(r1) } \quad (\text{load word}) \text{ r2} = \text{str}.i; \\
\text{lh } &\text{ r3, 4(r1) } \quad (\text{load half}) \text{ r3} = \text{str}.j; \\
\text{lbu } &\text{ r4, 6(r1) } \quad (\text{load byte unsigned}) \text{ r4} = \text{str}.c; \\
\end{align*}
\]

Displacement, continued.

Useful in ISAs without direct addressing and short immediates.

\[
\begin{align*}
! \text{lw } \text{r1, } (0x12345678) & \quad \text{Alas, no such instruction in DLX.} \\
\text{lh } \text{r2, } 0x1234 & \quad \text{Load high part of } \text{r2}: \text{r2} = 0x12340000 \\
\text{lw } &\text{ r1, 0x5678(r2) } \quad \text{r1} = \text{MEM}[0x5678+r2] = \text{MEM}[0x12345678] \\
\end{align*}
\]

ISA design parameter: displacement size.

Included in most ISAs.

Indexed Addressing:
Effective address is sum of two registers.

\[
\text{Load r4, (r1+r2) } \quad \text{r4} = \text{MEM}[\text{r1} + \text{r2}] \\
\]

Useful for array access. (\text{r1} address of first element.)

Included in most ISAs.

Memory Indirect Addressing:
Address of effective address is in register.

\[
\text{Load r1, @(r3) } \quad \text{r1} = \text{MEM}[\text{MEM}[\text{r3}]] \\
\]

Useful for dereferencing: \( i = *\text{ptr} \)

Included in some ISAs.

Others omit it since two loads would be as fast.
Autoincrement Addressing:
Perform register indirect access, then add constant to register.

Load r1,(r2)+ ! r1 = MEM[ r2 ]; r2 = r2 + 1

Useful for loops.
Included in some ISAs.

Autodecrement:
Subtract constant from register then perform register indirect access.

Load r1,-(r2) ! r2 = r2 - 1; r1 = MEM[ r2 ];

Useful for loops.
Included in some ISAs.

Scaled:
Effective address is constant1 + reg1 + reg2 * constant2.

Load r1,100(r2)[r3] ! r1 = MEM[ 100 + r2 + r3 × d ]

Useful for array access.
Included in some ISAs.

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.

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

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.

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

Smaller values used more frequently.
Immediate Sizes in VAX Code

![Graph showing immediate sizes in VAX Code]

**FIGURE 2.9 The distribution of immediate values is shown.**

Register Operands

Registers can be specified explicitly in instructions...

\[ \text{add } r1, r2, r3 \]

... but some ISAs allow them to be specified implicitly...
... that is, there is no need to specify a register number for some operands...
... because the instruction will always use a particular register.

Two Common Cases

**Accumulator:** A register for holding results.

**Stack:** A set of registers (and memory).

Accumulator

ISA specifies a special accumulator register...
... for example, \( ra \).

Arithmetic instructions use accumulator for destination and for one source operand.

For example: \( \text{add } r4 \), \( ra = ra + r4 \)

Advantage: Smaller instruction coding possible.

Disadvantage: “Extra” instructions needed to move data in and out of accumulator.

Stack

Registers organized as stack. Stack may extend into memory.

Most instructions read top one or two elements.

Example: Use register names: \( r1, r2, r3 \), with \( r1 \) top of stack, etc.

\[ \begin{array}{c}
\text{Before: } r1 = 1, r2 = 2, r3 = 3, r4 = 4, r4 = 8 \\
\text{add} \quad ! \text{Pop top two elements off stack, add, push sum on stack.} \\
\text{After: } r1 = 3, r2 = 4, r3 = 8
\end{array} \]

Special *Stack Machine* Instructions

\( \text{push } (\text{addr}) \) Read memory at \( (\text{addr}) \) and push on stack.

\( \text{pop } (\text{addr}) \) Pop data off stack and write to memory at \( (\text{addr}) \).

Advantage: Very short instructions possible.

Disadvantage: Some code requires extra instructions.