Instruction Set (ISA) Design and Addressing Modes

03-1

Material from sections 2.1, 2.2, and 2.3.

Outline

ISA Design Choices

It's more than just picking instructions.

Example: Easy ISA Design

See the big picture before being inundated with details.

ISA Design Choice Details

Screw up, and you'll be cursed for decades.

03-2

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

03-2

03-3

Easy ISA Design

Goal: Design easy-to-understand but not-necessarily-good ISA.

Use ISA design steps above.

I.A., Data Types

Hmmm.... 8-bit integer, 16-bit integer,..., 256-bit complex double,...

To keep it simple just use two types:

64-bit integer

64-bit floating-point

EE 4720 Lecture Transparency. Formatted 13:30, 26 January 2000 from lsli03.

03-3

I.B., Memory and Register Organization

03-4

First, Memory: The Questions

What *address space* size? That is, how many possible memory addresses?

What character size? That is, how many bits per location?

The Answers

Choose common address space size: 64 bits or $2^{64} = 18446744073709551616$[1] locations.

Note: implementation does not need to provide that much memory!!

Choose common character size: 8 bits.

Comments

Address space size usually matches fastest integer size.

Data type sizes should be multiple of character size ...
... *e.g.* 64-bit integer equals 8 characters.

[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.

03-5

I.B., Memory and Register Organization, Continued

Second, Registers

How many?

Choose 128. (Many systems have 64.)

What size?

They should be sized for the data types, so 64 bits.

Restrictions

Could restrict floats to *so-called* floating-point registers.

Could specify a special index register for memory addressing.

But we won't. Any register can be used for anything!

03-6

I.C., Addressing Modes

In case anyone forgot...

Addressing Mode: method used by an instruction to find source (operand) values and where to put destination (operand) value.

Consider:

```
add r1, [r2+4], 12  ! r1 = Mem[r2+4] + 12
```

Destination operand, `r1`, indicates result written to register `r1`.

First source operand, `[r2+4]`, in memory at address `r2+4`.

Second source operand, `12`, is part of instruction.

03-7

I.C., Addressing Modes, Continued

Data can be located in the instruction, a register, or memory.

If data in instruction (*immediate* data) we need to choose:

Type of immediate. (Usually a signed or unsigned integer.)

Size of immediate. (Big ones force design compromises.)

If data in register, not much to choose. (For simple design.)

If data in memory, need to specify how address computed.

03-8

I.C., Addressing Modes, Continued

For immediate addressing choose 20-bit signed integers.

It's possible to use more than one size or type, but we won't.

For memory addressing use two modes in Easy ISA:

Displacement: address is register value plus 64-bit constant (displacement).

Register indirect: address is register value.

Examples

```
add  r1, r1, 10      ! Last operand immediate.
add  r2, r2, 10.3    ! ERROR: Immediate should be integer.
add  r3, r3, [r4+20] ! Displacement addressing.
add  r5, r5, [r4]    ! Register indirect.
```

03-9

I.C., Addressing Modes, Continued

Need to choose which instructions use which addressing modes.

For simplicity, use addressing mode wherever it makes sense.

Discussed further below.

03-10

II.A., Data Movement Instructions

These instructions move data between registers to memory.

Often provided by special `load` and `store` instructions.

To keep things simple use arithmetic instructions for data movement:

```
add r1, [r2], 0  ! r1 = Mem[r2] + 0  (Don't need a load instruction.)
add [r2], r3, 0  !  Mem[r2] = r3 + 0  (Don't need a store instruction.)
```

II.B., Arithmetic and Logical Instructions.

Hmmm... `add`, `sub`, ... `cos`, `harcsin`, `gamma`,...

To keep things simple choose integer and float add and subtract:

`add` ⟨**desto**⟩ ⟨**s1**⟩ ⟨**s2**⟩ (Integer add.)

`addf` ⟨**desto**⟩ ⟨**s1**⟩ ⟨**s2**⟩ (Floating-point add.)

`sub` ⟨**desto**⟩ ⟨**s1**⟩ ⟨**s2**⟩ (Integer subtract.)

`subf` ⟨**desto**⟩ ⟨**s1**⟩ ⟨**s2**⟩ (Floating-point subtract.)

... where ...
... ⟨**desto**⟩ uses register or any memory addressing mode ...
... and ⟨**s1**⟩ and ⟨**s1**⟩ use any addressing mode.

03-12

II.B., Control Transfer Instructions (CTI).

CTI refers to jumps, branches, procedure calls, returns.

To keep things simple in Easy ISA, one CTI:

`b` ⟨**cond**⟩ ⟨**return**⟩ ⟨**target**⟩...
... where ⟨**cond**⟩ is a register, if contents non-zero branch taken ...
... ⟨**return**⟩ is a register, `PC` placed in ⟨**return**⟩ if branch taken ...
... ⟨**target**⟩ is a 64-bit immediate, the address to branch to.

Notes

This single instruction can implement procedure calls and returns.

The size of the ⟨**target**⟩ field is unrealistic.

03-13

II.B., Control Transfer Instructions (CTI).

Example:

```
sub r1, r2, 123         ! Put branch condition in r1
b   r1, r120, 0x12345   ! If r1 != 0 branch to 0x12345 and
                         put return address in r120.
```


03-13

03-14

II.C., Process Control Instructions

Don't include any, assume programmers honest and don't make mistakes.

03-15

III Instruction Coding

Goal: Determine binary format for instructions.

Steps

1 Determine which instructions similar, place into groups.

2 Assign numbers to opcodes so group-mates share a prefix (first few bits).

3 If necessary, assign numbers to operand types and addressing modes.

4 Decide where to place operand fields.

III Instruction Coding, continued

Step 1, divide into groups.

Two groups, arithmetic and branch:

Arithmetic instructions in one group because all have 3 operands and use the same addressing modes.

Step 2, assign numbers (codes) to opcodes.

Instruction	Group	Coding (Binary)
`b`	Branch	1
`add`	Arithmetic	000
`addf`	Arithmetic	001
`sub`	Arithmetic	010
`subf`	Arithmetic	011

Note: the first bit of the coding identifies group.

Note: no room to add new opcodes. (Bad).

03-17

III Instruction Coding, continued

Step 3, if necessary, assign numbers to operand types and addressing modes.

It's not necessary for the branch because only one possible type for each operand.

It is necessary for the arithmetic instructions.

Type (Addr. Mode)	Type Coding	Operand Contents	Operand Size
Register	00	Register num.	7
Immediate	01	20 bit signed int.	20
Displ.	10	64 bit and reg. num.	71
Reg. Indirect	11	Register num.	7

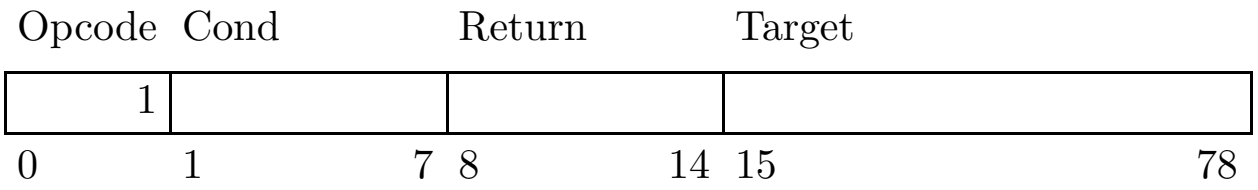
03-18

Step 4, decide where to place operand fields.

For decodability, operand type must precede operand.

For simplicity, put in same order as instruction.

Format for branch instruction (the easy one):



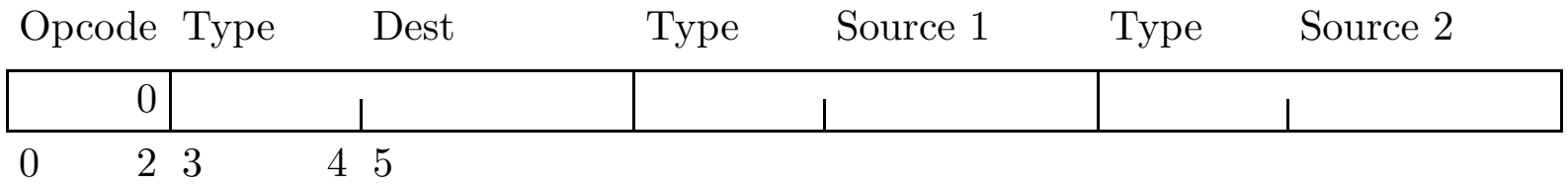
Note that size of this instruction fixed.

03-18

03-19

Step 4, decide where to place operand fields, continued.

Format for arithmetic instructions:



Note that size can vary.

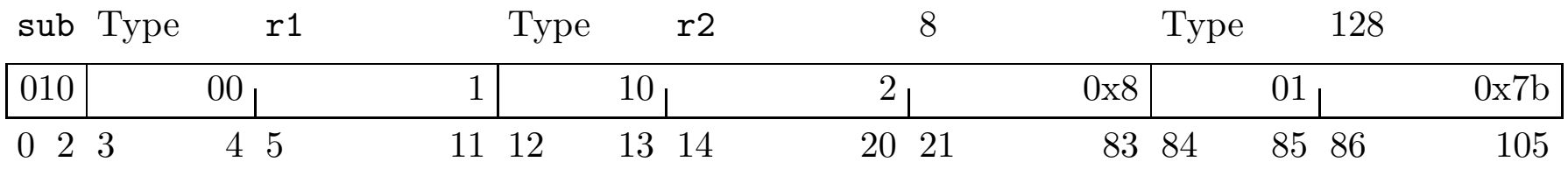
Cannot determine end of "Dest" field and following fields without knowing type.

EE 4720 Lecture Transparency. Formatted 13:30, 26 January 2000 from lsli03.
03-19

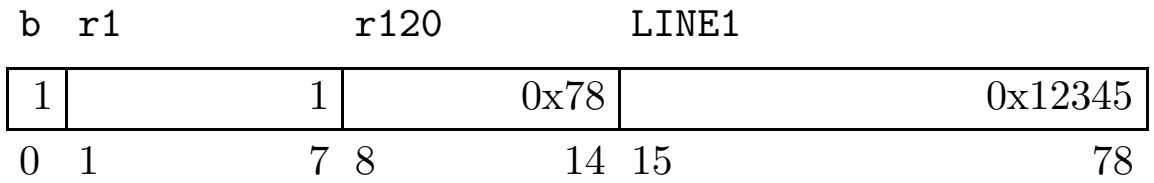
03-20

Instruction Examples

```
LINE1:                      ! LINE1 = 0x12345
 sub r1, [r2+8], 123        ! Put branch condition in r1
 b   r1, r120, LINE1        ! If r1 != 0 branch to 0x12345 and
                            ! put return address in r120.
```



Instruction length 106 bits or 13 bytes plus 2 bits, 6 bits go unused.



Note: $120_{10} = 78_{16}$.

Instruction length 79 bits or 7 bytes plus 7 bits, 1 bit goes unused.

03-21

Easy ISA Summary

64-bit integer

64-bit floating-point

64-bit address space.

8-bit characters.

128, 64-bit registers.

Immediate addressing using 20-bit signed integers.

Displacement addressing with 64-bit offsets.

Register indirect addressing.

Arithmetic Instructions: `add`, `addf`, `sub`, `subf`.

CTI Instruction: `b`.

03-22

Critique of Easy ISA

Major, indisputable, problems with Easy ISA:

Many useful instructions omitted and no room to add new ones.

Branch instruction too large (because of large target address).

Constant for indexed addressing too large (64 bits), wasting size.

Instructions may not be multiple of byte, wasting space.

The hardware to handle 64-bit offsets (for indexed addressing) ...
... could be used to handle 64-bit immediates ...
... but ISA specifies 20-bit immediates. (Why not add a second immediate size.)

No behavior specified for immediate used as a destination.

Cannot branch to an address held in a register (without using self-modifying code).

03-23

Additional problems from 1990's perspective:

Instruction size varies, difficult to "fetch ahead".

Arithmetic instructions access memory, complicating implementation.

ISA Design Choice Details

03-24

Outline

Data Types

Memory and Register Organization

ISA Classification

Addressing Modes

Displacement and Immediate Sizes

Data Types

03-25

To include a new data type:

Determine its size.

Define operations.

Add new instructions to operate on it.

03-26

Data Types for Simple 32-bit Machine

Type	Special Instructions
32-bit signed integer	
32-bit unsigned integer	`addu`
16-bit signed integer	`lh`, load half-word.
16-bit unsigned integer	`lhu`, load half-word unsigned.
8-bit signed integer	`lb`, load byte.
8-bit unsigned integer	`lbu`, load unsigned byte.
32-bit float	`addf`, add 32-bit floating-point.
64-bit float (double)	`addd`, add 64-bit floating-point.

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`, `lhu`, `lb`, and `lbu` instructions ...
... place data in low portion of 32-bit registers ...
... and place zeros or a sign bit high portion.

03-27

Data Type Tradeoffs

Possible benefit of a new data type.

Using one of the new instructions faster than many old instructions.

Possible drawbacks of a new data type.

Execution not much faster because ...
... data type is used infrequently or ...
... execution using other instructions nearly as fast.

More performance would be obtained if chip area used for new instructions was used elsewhere.

03-28

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:

`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**⟩.

03-30

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.

03-31

Overview of Popular Data Types

Common Data Type Sizes

Type usually specified with a size.

- *Byte, char, octet.* 1 byte (8 bits here).
- *Half word.* 2 bytes.
- *Word.* 4 bytes.
- *Doubleword.* 8 bytes.
- *Quadword.* 16 bytes.

03-32

Common Types and Sizes

- *Unsigned integer* and *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.

- *Floating-point.* Word, doubleword, quadword.

  Most newer machines use the IEEE 754 format.

03-33

- *Packed BCD.* Word, etc.

  Each word holds several BCD digits of a 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.

03-34

Integer, Packed BCD, and Packed Integer Example

Consider integers 1239_{10} and 5678_{10}.

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

... 1239_{10} = `0x04d7` =

Sign		Short Int.	
0		0x04d7	
31	16	15	0

(in 32-bit register) and ...

... 5678_{10} = `0x162e` =

Sign		Short Int.	
0		0x1628	
31	16	15	0

As packed BCD integers:

1239_{10} = `0x1239` =

MSD							LSD
0	0	0	0	1	2	3	9
31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0

5678_{10} = `0x5678` =

MSD							LSD
0	0	0	0	5	6	7	8
31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0

EE 4720 Lecture Transparency. Formatted 13:30, 26 January 2000 from lsli03.

03-34

03-35

Consider two lists of integers: $\{1, 2, 3, 9\}$ and $\{5, 6, 7, 8\}$.

As packed 4-bit unsigned integer (8 4-bit numbers per word):

$\{1, 2, 3, 9\} =$ `0x1239` $=$

0	0	0	0	1	2	3	9
31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0

$\{5, 6, 7, 8\} =$ `0x5678` $=$

0	0	0	0	5	6	7	8
31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0

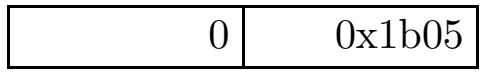
EE 4720 Lecture Transparency. Formatted 13:30, 26 January 2000 from lsli03.
03-35

03-36

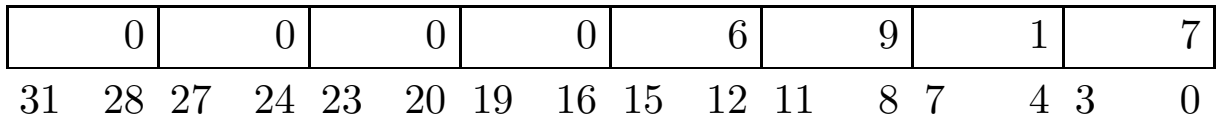
Examples of Packed Data Types

Addition:	Integer	Packed BCD	Packed Int.
	`0x04d7`	`0x1239`	`0x1239`
	`+ 0x162e`	`+ 0x5678`	`+ 0x5678`
	`0x1b05`	`0x6917`	`0x68af`
	$= 6917_{10}$	$= 6917_{10}$	$=\{6, 8, 10, 15\}$

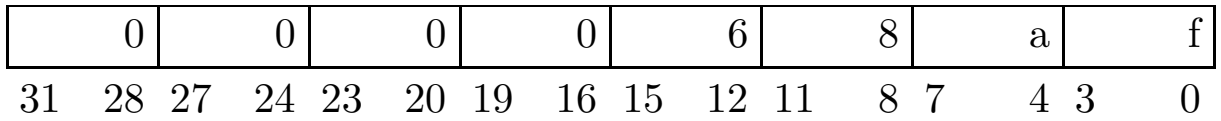
`0x1b05`=



`0x6917` =



`0x68af` =



Addition of packed integers is *saturating*: ...

... result is maximum value if sum exceeds maximum value.

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

03-36

03-37

Data Type Usage

Data below for SPEC92 programs on VAX.

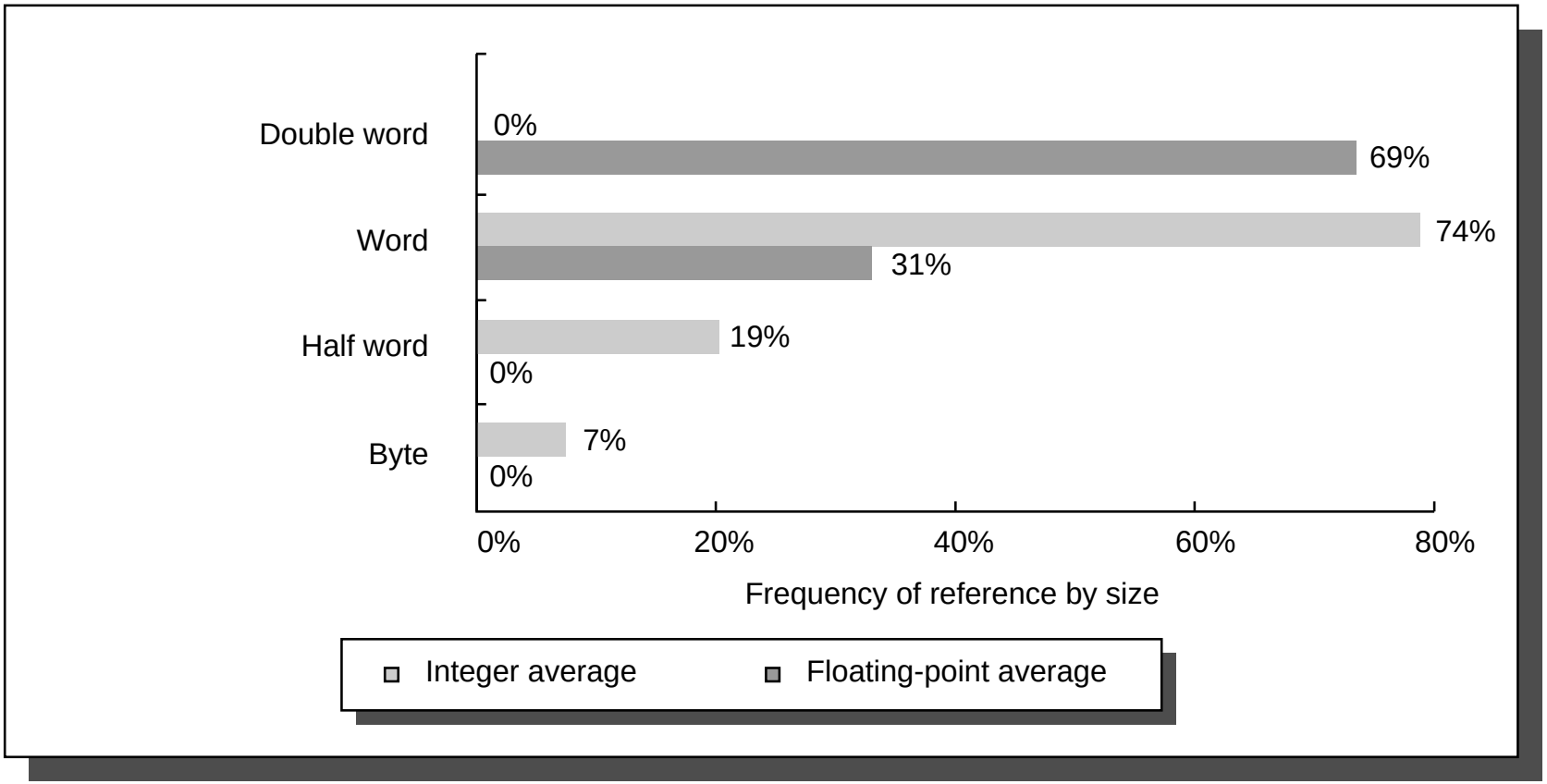


FIGURE 2.16 Distribution of data accesses by size for the benchmark programs.

EE 4720 Lecture Transparency. Formatted 13:30, 26 January 2000 from lsli03.

03-37

03-38

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.

03-39

Data Type Coding

How data 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

03-40

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

Operands ⟨**op1**⟩ and ⟨**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.

03-41

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

03-42

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.

03-43

Memory

Stores code and data.

Simple to programmer ... despite complex implementation.

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

$2^{128} = 340282366920938463463374607431768211456$ [2] 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.

[2] Three hundred forty undecillion, two hundred eighty two decillion, three hundred sixty six nonillion, 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 seven trillion, four hundred thirty one billion, seven hundred sixty eight million, two hundred eleven thousand, four hundred fifty six.

Memory Addressing

03-44

Address Interpretation

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

Size of sequence depends upon instruction.

E.g., DLX `lw`, load word, instruction reads four bytes.

E.g., DLX `lb`, load byte, instruction reads one byte.

Example:

```
lw  r1, 0(r2)  ! Load r1 with 4 bytes starting at addr. in r2.
lb  r3, 0(r2)  ! Load r3 with byte at address in r2.
! Register r1 = r3 if  r1 < 128 and r1 > 0.
```

03-45

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.
```


EE 4720 Lecture Transparency. Formatted 13:30, 26 January 2000 from lsli03.
03-45

03-46

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.

03-47

Immediate

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

03-48

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

Effective address is a constant.

```
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.

03-49

Register Deferred or *Register Indirect*

Effective address in register.

```
Load r4, (r1)       ! r4 = MEM[r1]
add  r4, r2, (r1)   ! r4 = r2 + MEM[r1]
```

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

Included in most ISAs.

03-50

Displacement

Effective address is register plus constant.

```
Load r4, 100(r1) !  r4 = MEM[ r1 + 100 ]
```

Useful for accessing elements of a structure:

```
! In c: struct { int i; short int j; unsigned char c; } str;
! r1 = &str;
lw  r2, 0(r1)    ! (load word) r2 = str.i;
lh  r3, 4(r1)    ! (load half) r3 = str.j;
lbu r4, 6(r1)    ! (load byte unsigned) r4 = str.c;
```

03-51

Displacement, continued.

Useful in ISAs without direct addressing and short immediates.

```
! lw r1, (0x12345678)  ! Alas, no such instruction in DLX.
 lhi r2, #0x1234       ! Load high part of r2: r2 = 0x12340000
 lw  r1, 0x5678(r2)    ! r1 = MEM[0x5678+r2] = MEM[0x12345678]
```

ISA design parameter: displacement size.

Included in most ISAs.

03-52

Indexed

Effective address is sum of two registers.

```
Load r4, (r1+r2) !  r4 = MEM[ r1 + r2 ]
```

Useful for array access. (`r1` address of first element.)

Included in most ISAs.

Memory Indirect

Address of effective address is in register.

```
Load r1,@(r3) !  r1 = MEM[ MEM[ r3 ] ].
```

Useful for dereferencing: `i = *ptr`

Included in some ISAs.

Others omit it since two loads would be as fast.

03-53

Autoincrement

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.

03-54

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

03-55

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.

03-56

Usage of Addressing Modes

Do we really need all those addressing modes?

Memory Addressing Usage in VAX Code.

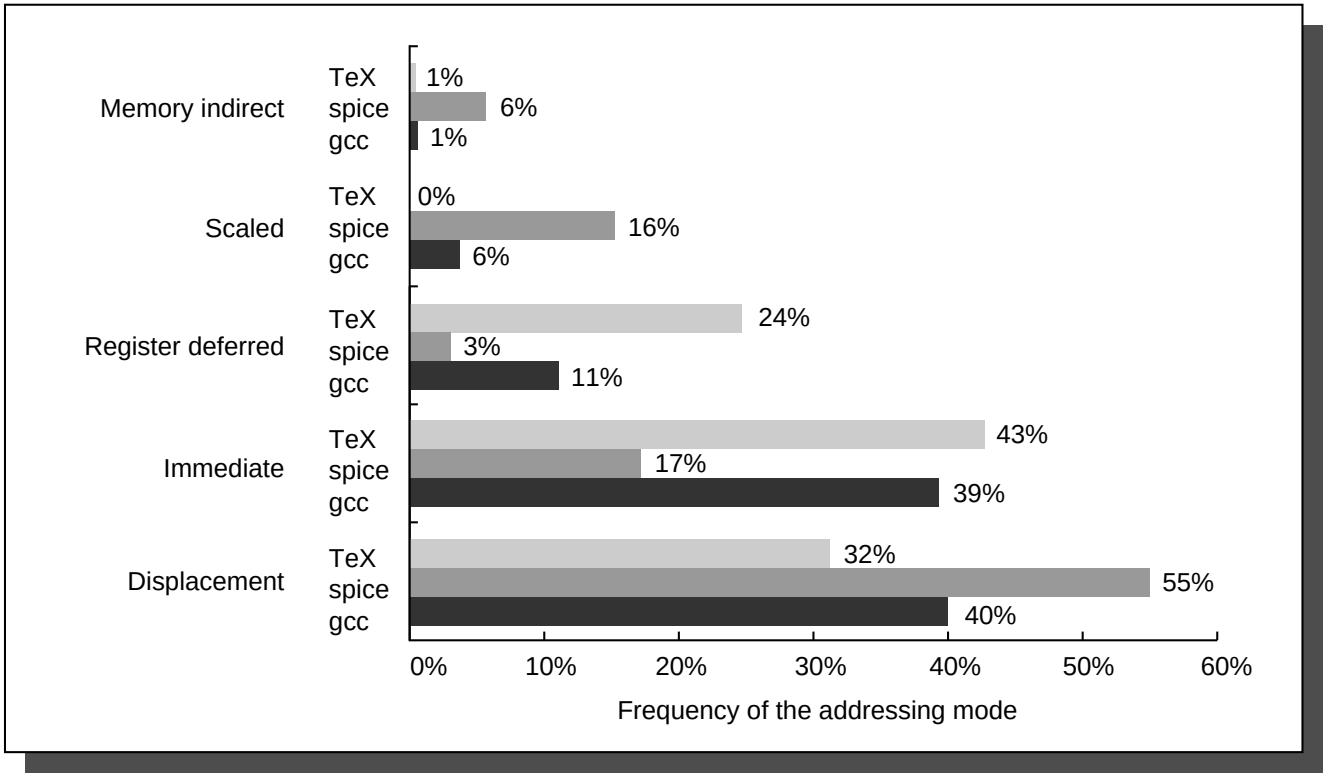


FIGURE 2.6 Summary of use of memory addressing modes (including immediates).

03-56

03-57

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.

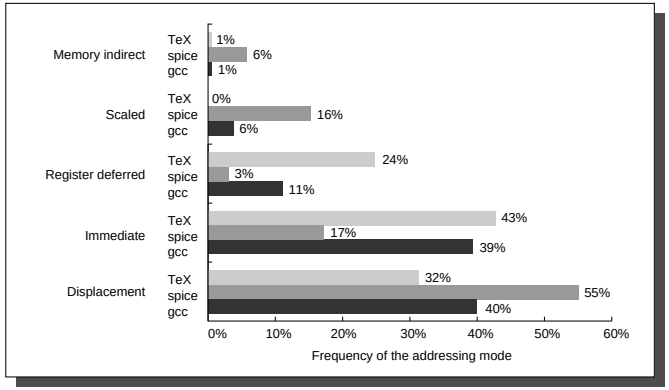


FIGURE 2.6 Summary of use of memory addressing modes (including immediates).

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).

03-58

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.

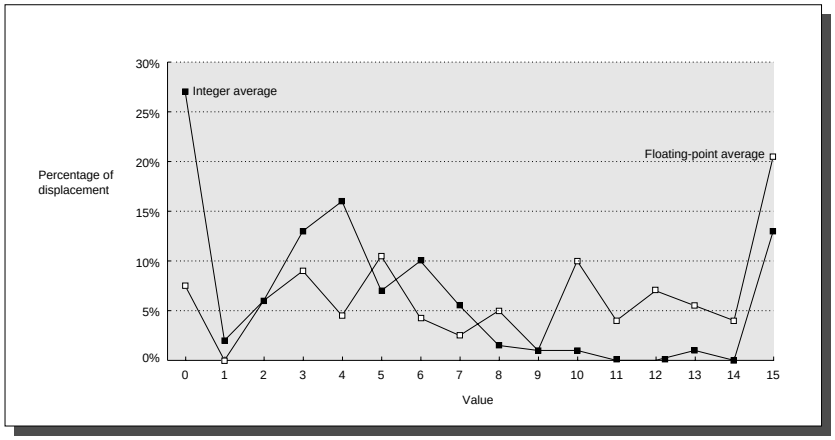


FIGURE 2.7 Displacement values are widely distributed.

03-58

03-59

Displacement Size in SPECint92 and SPECfp92 Programs on MIPS.

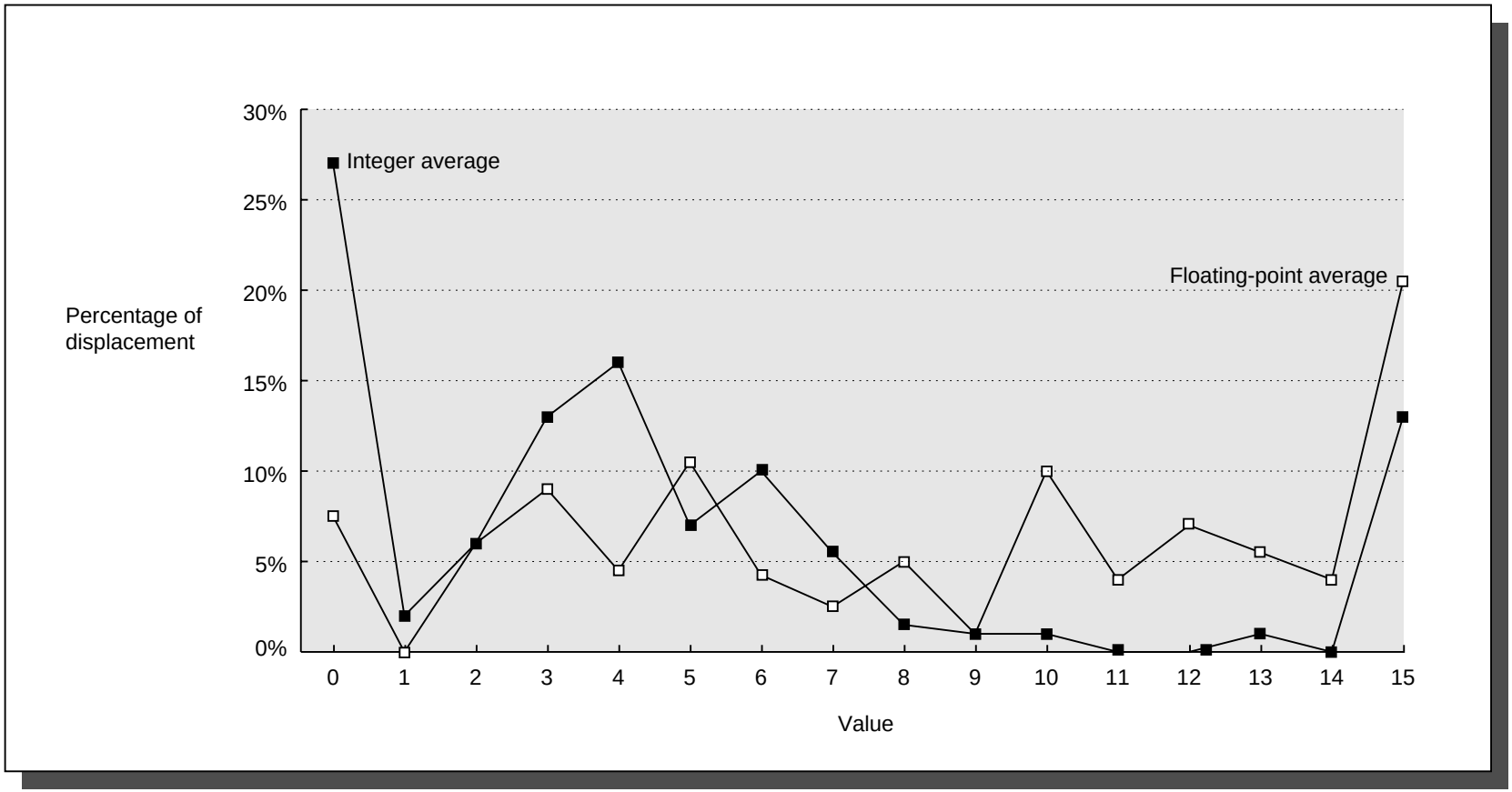


FIGURE 2.7 Displacement values are widely distributed.

EE 4720 Lecture Transparency. Formatted 13:30, 26 January 2000 from lsli03.

03-59

03-60

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.

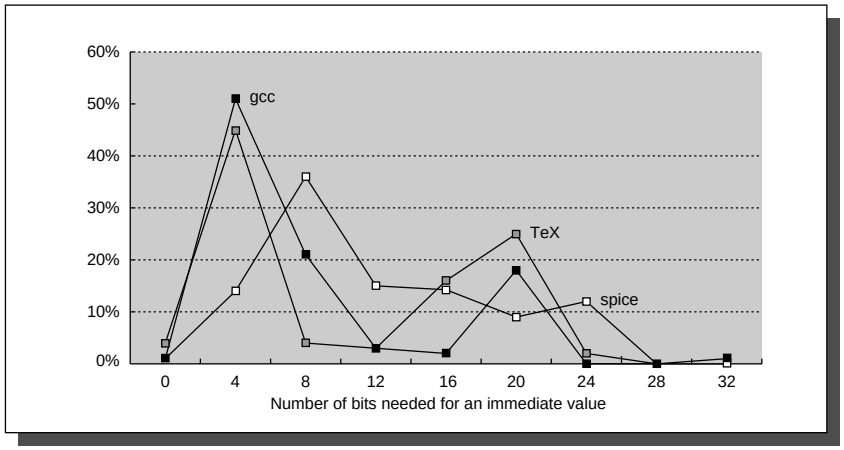


FIGURE 2.9 The distribution of immediate values is shown.

EE 4720 Lecture Transparency. Formatted 13:30, 26 January 2000 from lsli03.
03-60

03-61

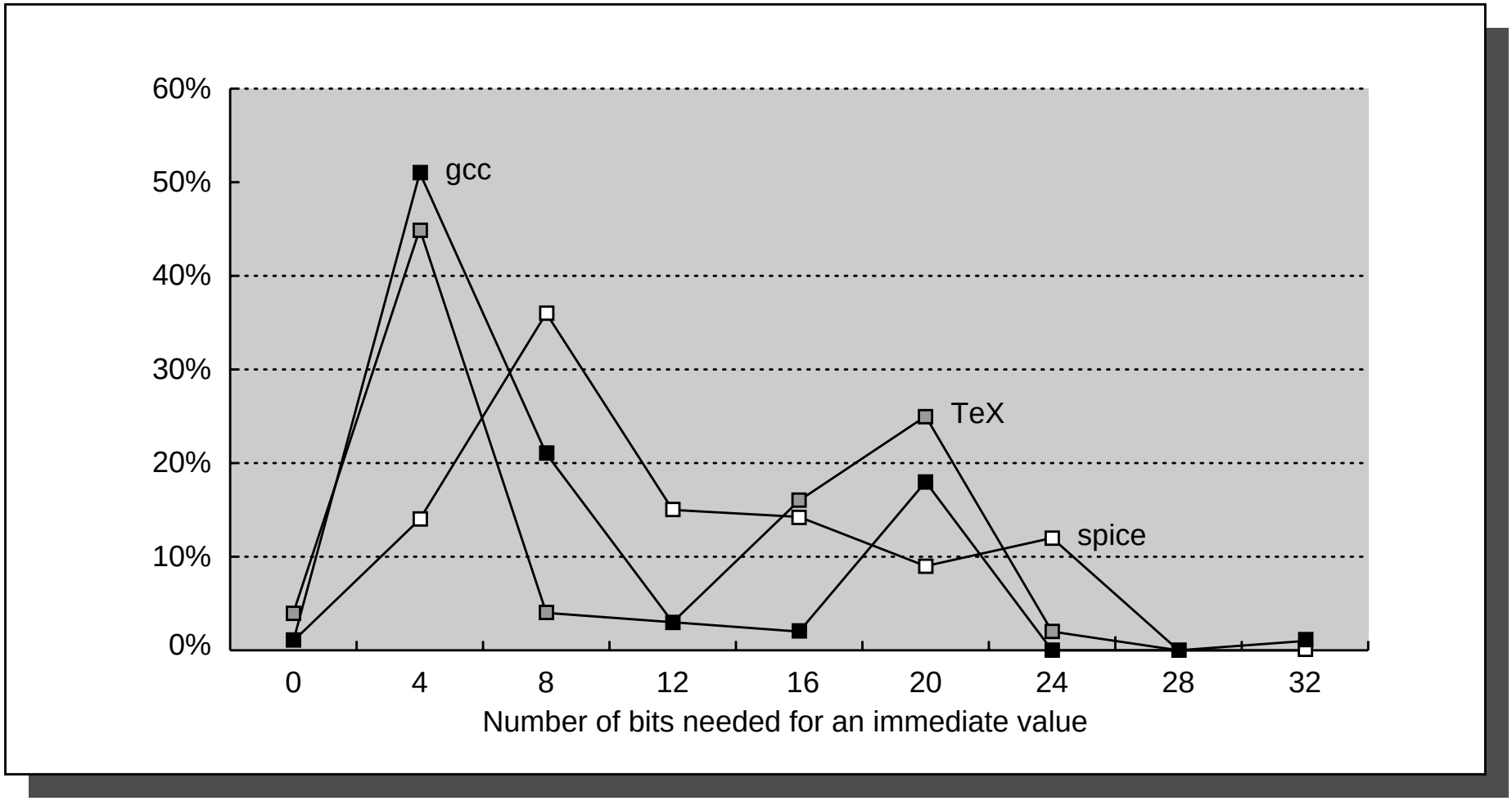


FIGURE 2.9 The distribution of immediate values is shown.

EE 4720 Lecture Transparency. Formatted 13:30, 26 January 2000 from lsli03.
03-61

03-62

Specifying Register Operands

Registers can be specified explicitly in instructions ...

```
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).

03-62

03-63

Accumulator

ISA specifies a special accumulator register . . .
. . . for example, `ra`.

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

For example: `add r4 !ra = ra + r4`

Advantage: Smaller instruction coding possible.

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

03-64

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.

```
! Before r1 = 1, r2 = 2, r3, = 4, r4 = 8
add   ! Pop top two elements off stack, add, push sum on stack.
! After r1 = 3, r2 = 4, r3, = 8
```

Special *Stack Machine* Instructions

`push` ⟨**addr**⟩ Read memory at ⟨**addr**⟩ and push on stack.

`pop` ⟨**addr**⟩ Pop data off stack and write to memory at ⟨**addr**⟩.

Advantage: Very short instructions possible.

Disadvantage: Some code requires extra instructions.

03-65

Miscellaneous Variations

Operands per Instruction

Three typically used.

Two sometimes used.

Factors:

Instruction coding (bits to specify operands).

03-66

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.