

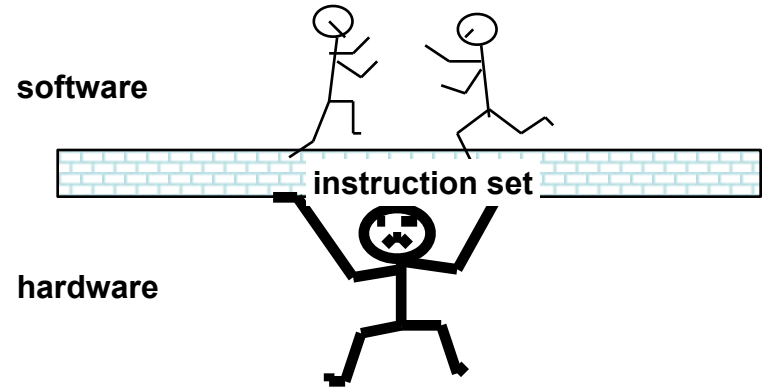
CPEN 411: Computer Architecture

Slide Set #3: Instruction Set Architecture Design

Instructor: Mieszko Lis

Original Slides: Professor Tor Aamodt

Instruction Set Architecture (ISA)



- The low-level software interface to the computer
 - Language the computer understands
 - Must translate any programming language into this language
 - Examples: ARM, x86, PowerPC, **MIPS**, SPARC, ...
- **ISA is the set of features “visible to programmer”**

Levels of Representation

High Level Language
Program

Compiler

Assembly Language
Program

Assembler

Machine Language
Program

Machine Interpretation

Control Signal
Specification

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

LW R15,0(R2)
LW R16,4(R2)
SW R16,0(R2)
SW R15,4(R2)

1000 1100 0110 0010 0000 0000 0000 0000
1000 1100 1111 0010 0000 0000 0000 0100
1010 1100 1111 0010 0000 0000 0000 0000
1010 1100 0110 0010 0000 0000 0000 0100

ALUOP[0:3] ← InstrReg[9:12] & MASK

-
-

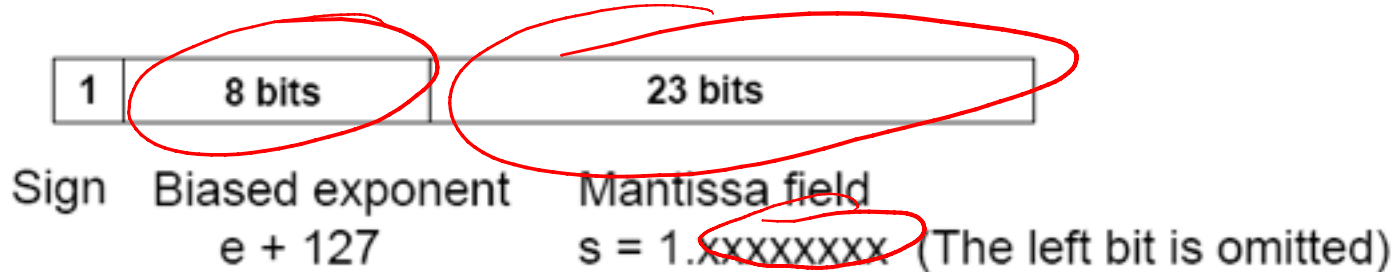
Review: Scientific Notation

Examples...

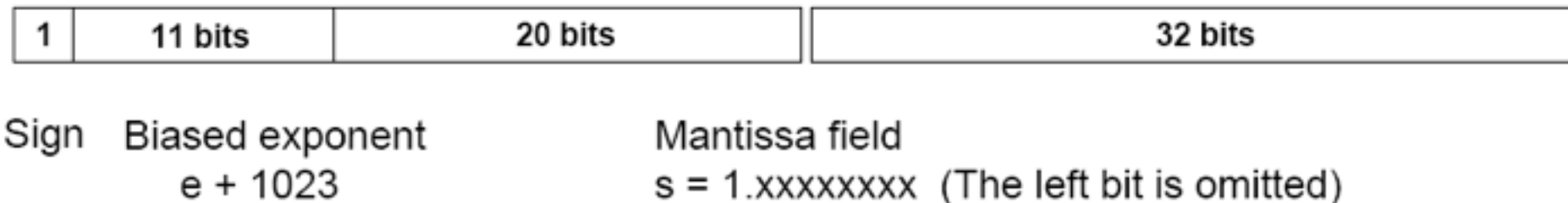
- 2.9979245×10^8 (speed of light, m/s)
- 6.0221413×10^{23} (Avogadro's number, mol⁻¹)
- $1.3806488 \times 10^{-23}$ (Boltzmann constant, J/K)

Review: Floating-Point

Single-Precision Format: (32 bits wide)



Double-Precision Format: (64 bits wide)



Smallest positive single-precision normalized number

0	00000001	000 . . .	0000
---	----------	-----------	------

$$1.000000000000000000000000_2 \times 2^{-126} \approx 1.2 \times 10^{-38}$$

Largest positive single-precision normalized number

0	11111110	111 . . .	1111
---	----------	-----------	------

$$1.111111111111111111111111_2 \times 2^{+127} \approx 3.4 \times 10^{38}$$

Example #1: Determine the IEEE 754 single-precision representation of -6.375

$$-6.375_{10} = -110.011_2 = -1.10011_2 \times 2^2$$

$$\text{Biased exponent} = 2 + 127 = 129_{10} = 10000001_2$$

1	10000001	100110000000000000000000
---	----------	--------------------------

Example #2: Determine what decimal number is represented by the following bits interpreted as a single-precision IEEE 754 value

0	01111100	0100	...	0
---	----------	------	-----	---

Value for Example 2 is:

A: $0.01_{10} \times 10^{1111100}$

B: $1.01_2 \times 2^{124}$

C: $1.01_2 \times 2^{-3}$

D: $0.01_2 \times 2^{-3}$

E: Not sure

Example #1: Determine the IEEE 754 single-precision floating-point representation of the decimal number -6.375.

$$-6.375_{10} = -110.011_2 = -1.10011_2 \times 2^2$$

$$\text{Biased exponent} = 2 + 127 = 129_{10} = 10000001_2$$

1	10000001	10011000000000000000000000000000
---	----------	----------------------------------

Example #2: Determine what decimal number is represented by the following bits interpreted as a single-precision IEEE 754 value

0	01111100	0100	...	0
---	----------	------	-----	---

Value for Example 2 is:

- A: $0.01_{10} \times 10^{1111100}$
- B: $1.01_2 \times 2^{124}$
- C: $1.01_2 \times 2^{-3}$
- D: $0.01_2 \times 2^{-3}$
- E: Not sure

A: $0.01_{10} \times 10^{1111100}$

B: $1.01_2 \times 2^{124}$

C: $1.01_2 \times 2^{-3}$

$$D: 0.01_2 \times 2^{-3}$$

E: Not sure

Example #1: Determine the IEEE 754 single-precision floating-point representation of the decimal value 10.5.

$$-6.375_{10} = -110.011_2 = -1.10011_2 \times 2^2$$

$$\text{Biased exponent} = 2 + 127 = 129_{10} = 10000001_2$$

1	10000001	10011000000000000000000000
---	----------	----------------------------

Example #2: Determine what decimal number is represented by the following bits interpreted as a single-precision IEEE 754 value

0	01111100	0100	...	0
---	----------	------	-----	---

Review: Hex Notation

- Often use “hex” notation (base 16 instead of base 10)
- Digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Example 1: $0xF = 15_{10} = 01111_2$
- Example 2: $0x12 = 18_{10} = 10010_2$

Example ISA: MIPS

- Use MIPS64 subset in examples in course
- Similar to “PISA” ISA used in assignments
- Syntax used in class, problem sets & tests is simplified versus used by compiler

MIPS Registers

32 general-purpose registers R0, R1, ... R31

- Each contains 64-bits (8 bytes)
- NOTE: Value of R0 is always 0
- Notation: Regs[R3] is 64-bit value in R3.

32 floating-point registers, F0, F1, ... F31

- Either a 32- bit or 64-bit floating-point number

Special Register: PC (program counter)

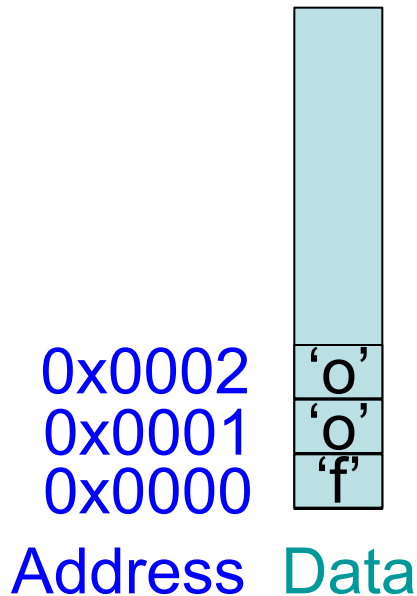
Memory

32 registers not enough to store data for useful programs.

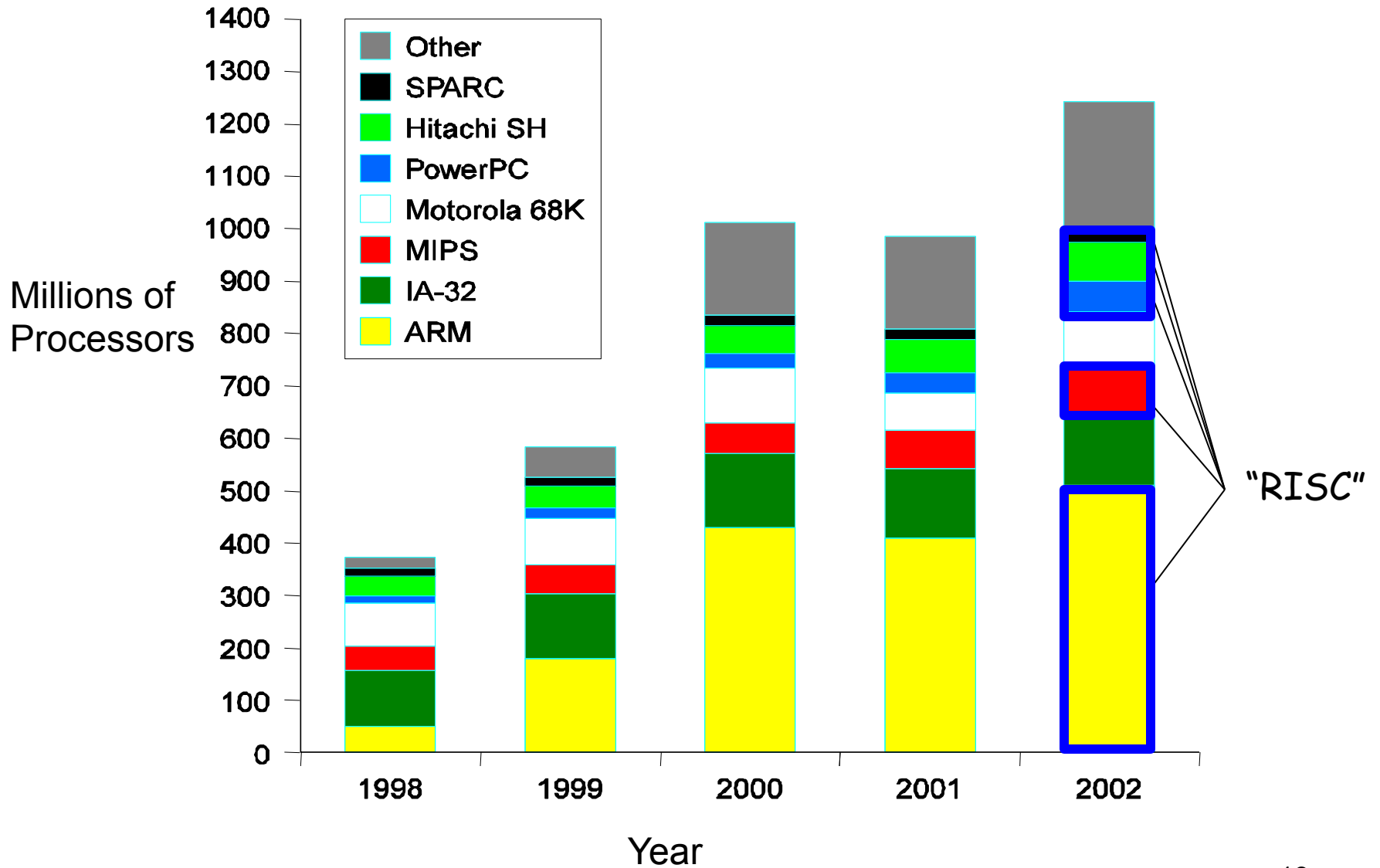
All computers have “random access memory” (RAM) that is used to store data.

To get data, need to know it's address.

Also store instructions in memory.



Which ISA?



Why MIPS?

- Today: MIPS still used in embedded microprocessors.
- History: Developed at Stanford (by current Stanford President). In fast microprocessors of mid-90's (e.g., MIPS R10000).
- MIPS is a Reduced Instruction Set Computer (RISC) ISA just like ARM, SPARC, PowerPC. Arguably simpler so more suitable when learning principles.

Why MIPS?

- MIPS64 is representative of all RISC architectures
 - i.e., if you understand one, learning another one is very easy
- x86 today -> uses RISC internally
 - (ANY ISA that isn't RISC would use RISC internally if you wanted high performance)

The x86 isn't all that complex—it just doesn't make a lot of sense.

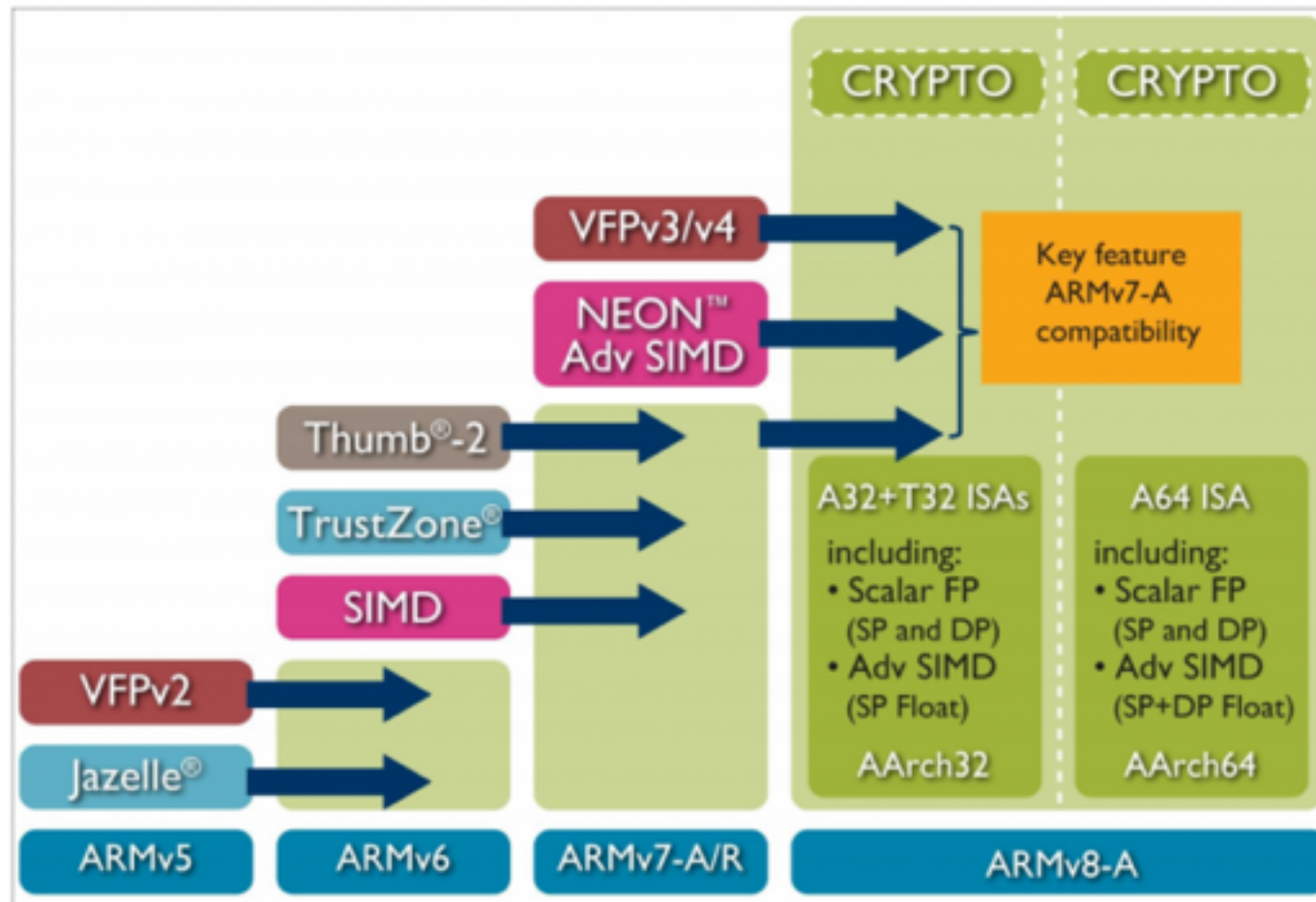
Mike Johnson

*Leader of 80x86 Design at AMD,
Microprocessor Report (1994)*

- RISC roots -> CDC 6600 and Cray-1
- 1st RISC microprocessors: IBM 801, Berkeley RISC, Stanford MIPS

Example 2: ARM

ARMv8-A – Context



ARM Instruction Encoding

ADD immediate

Encoding T1 All versions of the Thumb ISA.

ADD{S} <Rd>, <Rn>, #<imm3>

ADD{C} <Rd>, <Rn>, #<imm3>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

Encoding T2 All versions of the Thumb ISA.

ADD{S} <Rdn>, #<imm8>

ADD{C} <Rdn>, #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

Encoding T3 ARMv7-M

ADD{S}<C>{W} <Rd>, <Rn>, #<const>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	Rn			0	imm3			Rd			imm8									

Encoding T4 ARMv7-M

ADDW{C} <Rd>, <Rn>, #<imm12>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	Rn			0	imm3			Rd			imm8									

Elements of an ISA

- Set of machine-recognized data types
 - bytes, words, integers, floating point, strings, . . .
- Operations performed on those data types
 - Add, sub, mul, div, xor, move,
- Programmable storage
 - Registers, program counter, memory
- Methods of identifying and obtaining data referenced by instructions (addressing modes)
 - Addressing modes: Literal, register, absolute, relative, reg + offset, ...
 - Endianness, alignment restrictions
- Format (encoding) of the instructions
 - Op code, operand fields, ...

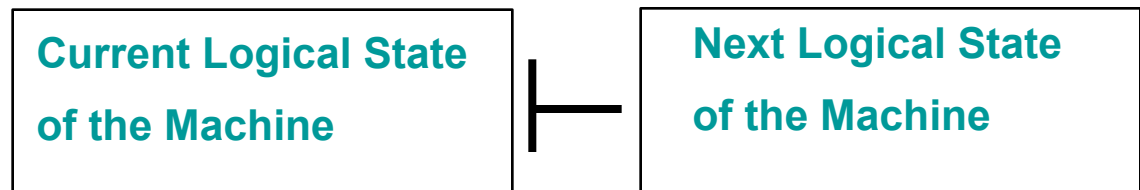
Elements of an ISA

- Set of machine-recognized data types
 - bytes, words, integers, floating point, strings, . . .
- Operations performed on those data types
 - Add, sub, mul, div, xor, move,
- Programmable storage
 - Registers, program counter, memory
- Methods of identifying and obtaining data referenced by instructions (addressing modes)
 - Addressing modes: Literal, register, absolute, relative, reg + offset, ...
 - Endianness, alignment restrictions
- Format (encoding) of the instructions
 - Op code, operand fields, ...

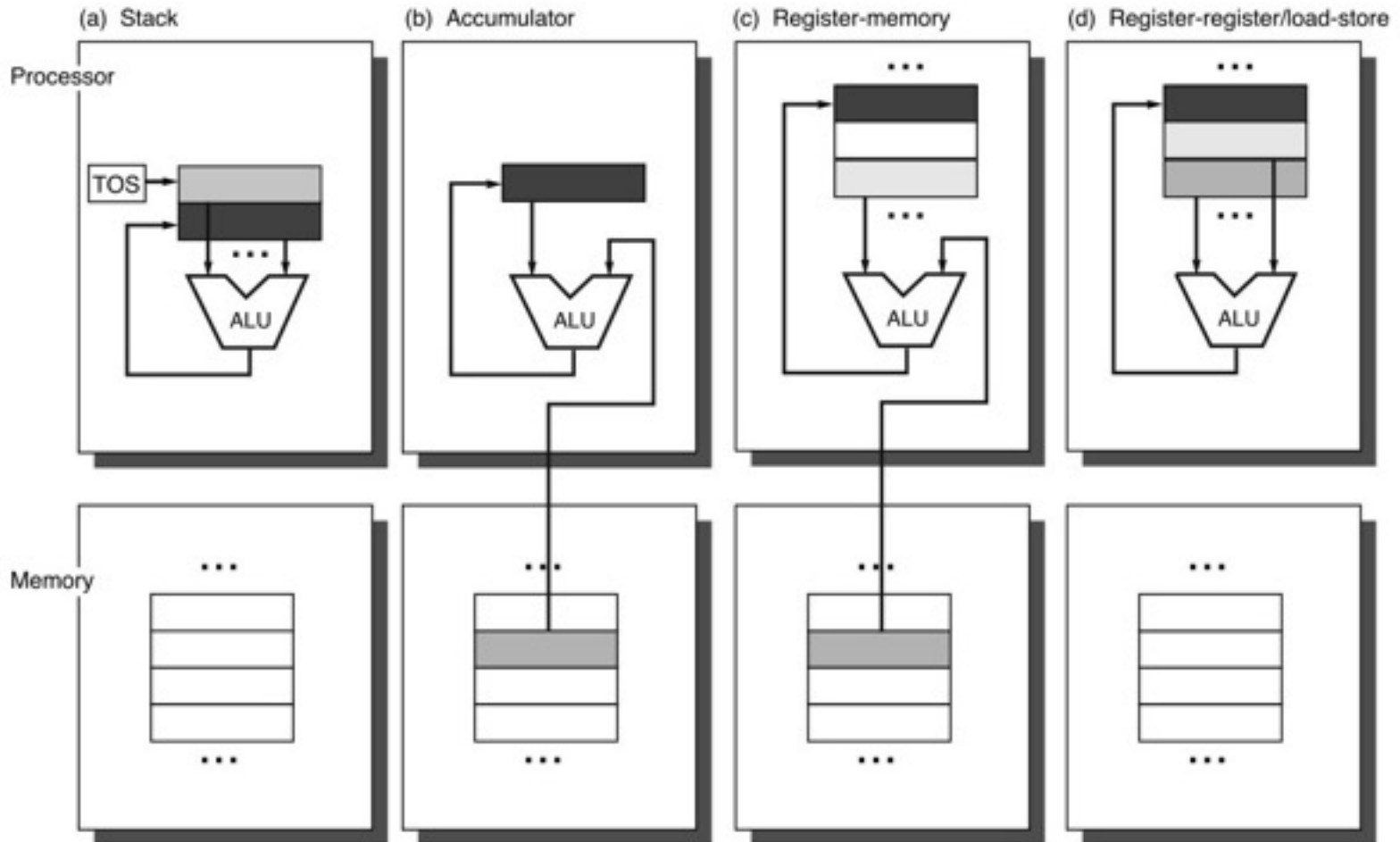
**Current Logical State
of the Machine**

Elements of an ISA

- Set of machine-recognized data types
 - bytes, words, integers, floating point, strings, . . .
- Operations performed on those data types
 - Add, sub, mul, div, xor, move,
- Programmable storage
 - Registers, program counter, memory
- Methods of identifying and obtaining data referenced by instructions (addressing modes)
 - Addressing modes: Literal, register, absolute, relative, reg + offset, ...
 - Endianness, alignment restrictions
- Format (encoding) of the instructions
 - Op code, operand fields, ...



Four ISA Classes



Four ISA Classes

How to compute " $C = A + B$ ":

Stack

```
Push A
Push B
Add
Pop C
```

Accumulator

```
Load A
Add B
Store C
```

Reg/Mem

```
Load R1, A
Add R3, R1, B
Store R3, C
```

Reg/Reg

```
Load R1, A
Load R2, B
Add R3, R1, R2
Store R3, C
```


Four IS

For the example below, which architecture do you think requires the fewest bits of storage to represent the assembly code?

A: Stack
B: Accumulator
C: Reg/Mem
D: Reg/Reg

How to compute " $C = A + B$ ":

Stack

```
Push A
Push B
Add
Pop C
```

Accumulator

```
Load A
Add B
Store C
```

Reg/Mem

```
Load R1, A
Add R3, R1, B
Store R3, C
```

Reg/Reg

```
Load R1, A
Load R2, B
Add R3, R1, R2
Store R3, C
```

Question

- Assume A, B, and C reside in memory. Assume opcodes are 8-bits. Memory addresses are 64-bits, and register addresses are 6-bits (64 registers)
- For each class of ISA, how many addresses or names appear in each instruction for the code to compute $C = A + B$, and what is the total code size?

Four ISA Classes

Stack

Push A
Push B
Add
Pop C

Accumulator

Load A
Add B
Store C

Reg/Mem

Load R1, A
Add R3, R1, B
Store R3, C

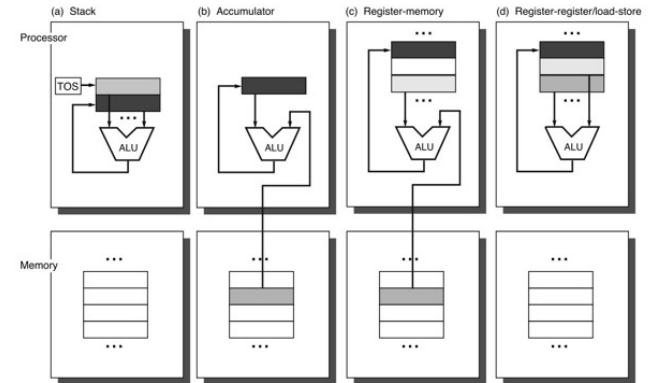
Reg/Reg

Load R1, A
Load R2, B
Add R3, R1, R2
Store R3, C

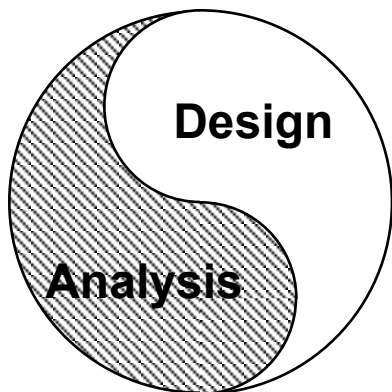
- Stack 3 addresses, total bits = $4 \cdot 8 + 3 \cdot 64$ = 224 bits
- Accumulator 3 addresses, total bits = $3 \cdot (8 + 64)$ = 216 bits
- Reg/Mem 3 addresses+4 regs, total bits = $3 \cdot 64 + 4 \cdot 6 + 3 \cdot 8$ = 240 bits
- Reg/Reg 3 addresses+6 regs, total bits = $3 \cdot 64 + 6 \cdot 6 + 4 \cdot 8$ = 260 bits

Trade-Offs

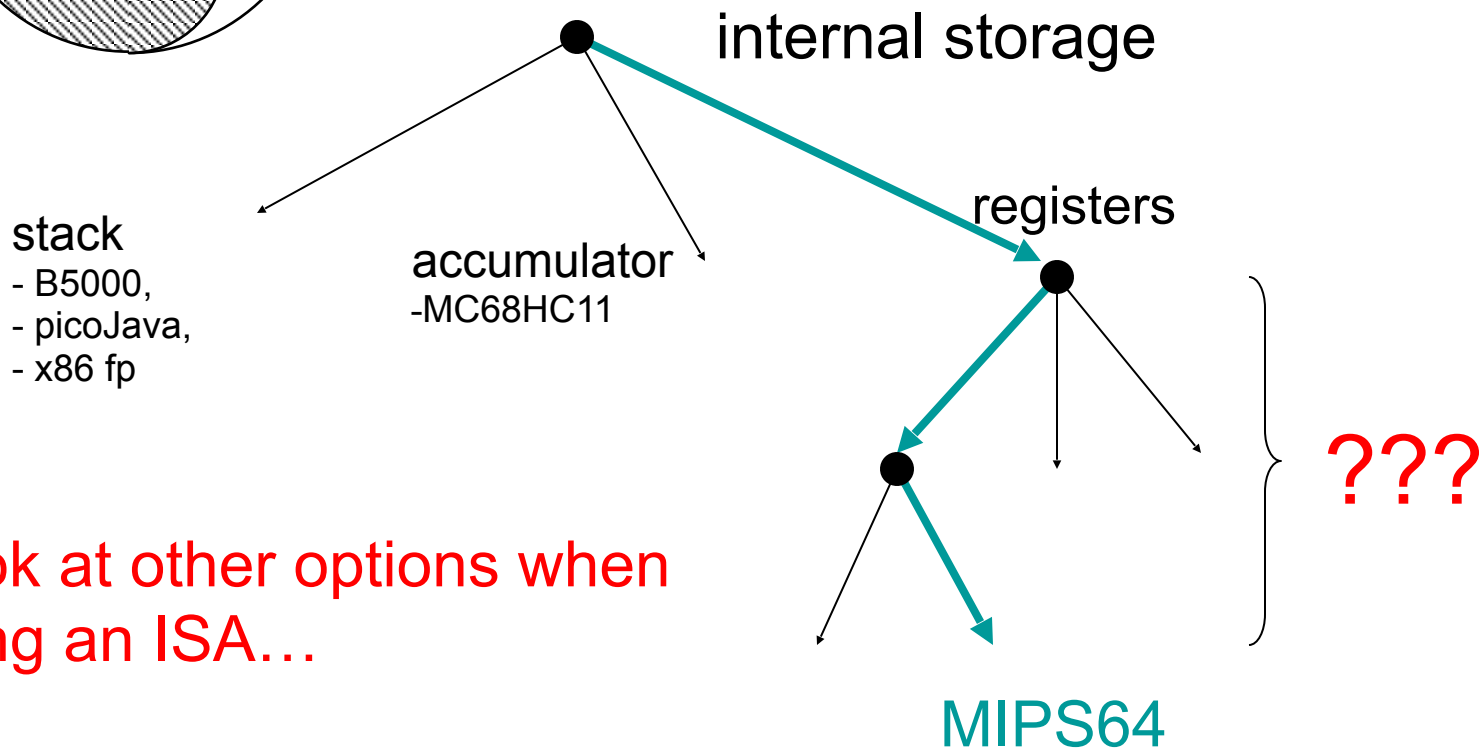
- Stack
 - extra code / memory bandwidth
 - hard to “reorder” instructions
 - + good code density
- Accumulator
 - difficult for compiler
 - extra code / memory bandwidth
 - + good code density
- Register-Memory
 - # of registers may be limited
 - operands not equivalent
 - + good code density
- Register-Register
 - poor code density (more instructions to do same thing)
 - + easy to pipeline
 - + easy for compiler to generate efficient code



Measurement and Evaluation



Architecture is an iterative process
-- searching the space of possible designs
-- at all levels of computer systems



Let's look at other options when designing an ISA...

Memory Addressing

Each data value is stored in memory at a location determined by a “memory address”.

Memory Addressing

Each data value is stored in memory at a location determined by a “memory address”.

- What does a memory address (and length) mean?

Memory Addressing

Each data value is stored in memory at a location determined by a “memory address”.

- What does a memory address (and length) mean?
 - Byte-addressable (desktop, notebook, server computers)
 - Address specifies a multiple of 8-bits (a byte)
 - Access power of 2 number of bytes (8-bits, 16-bits, 32-bits, 64-bits)

Memory Addressing

Each data value is stored in memory at a location determined by a “memory address”.

- What does a memory address (and length) mean?
 - Byte-addressable (desktop, notebook, server computers)
 - Address specifies a multiple of 8-bits (a byte)
 - Access power of 2 number of bytes (8-bits, 16-bits, 32-bits, 64-bits)
 - Word addressable (often found in embedded processors)
 - Word size = number of bits used in arithmetic operations such as addition (8, 16, 24-bits common word sizes for embedded processors)
 - Address specifies which “word” to access in memory.

Memory Addressing

Each data value is stored in memory at a location determined by a “memory address”.

- What does a memory address (and length) mean?
 - Byte-addressable (desktop, notebook, server computers)
 - Address specifies a multiple of 8-bits (a byte)
 - Access power of 2 number of bytes (8-bits, 16-bits, 32-bits, 64-bits)
 - Word addressable (often found in embedded processors)
 - Word size = number of bits used in arithmetic operations such as addition (8, 16, 24-bits common word sizes for embedded processors)
 - Address specifies which “word” to access in memory.
- When reading/writing multiple bytes of data in memory, which order are the bytes put in?
 - Little Endian: byte at “xxxxx000₂” is least significant
 - [7 6 5 4 3 2 1 0]
 - Big Endian: byte at “xxxxx000₂” is most significant
 - [0 1 2 3 4 5 6 7]

Memory Addressing

Each data value is stored in memory at a location determined by a “memory address”.

- What does a memory address (and length) mean?
 - Byte-addressable (desktop, notebook, server computers)
 - Address specifies a multiple of 8-bits (a byte)
 - Access power of 2 number of bytes (8-bits, 16-bits, 32-bits, 64-bits)
 - Word addressable (often found in embedded processors)
 - Word size = number of bits used in arithmetic operations such as addition (8, 16, 24-bits common word sizes for embedded processors)
 - Address specifies which “word” to access in memory.
- When reading/writing multiple bytes of data in memory, which order are the bytes put in?
 - Little Endian: byte at “xxxxx000₂” is least significant
 - [7 6 5 4 3 2 1 0]
 - Big Endian: byte at “xxxxx000₂” is most significant
 - [0 1 2 3 4 5 6 7]
- x86 => little endian
- SPARC/PowerPC => big endian
- MIPS/ARM/IA64/PA-RISC => mode bit

Memory Addressing

Each data value is stored in memory at a “memory address”.

- What does a memory address (address) specify?
 - Byte-addressable (desktop, notebook)
 - Address specifies a multiple of 8
 - Access power of 2 number of bytes
 - Word addressable (often found in embedded processors)
 - Word size = number of bits used in arithmetic operations such as addition (8, 16, 24-bits common word sizes for embedded processors)
 - Address specifies which “word” to access in memory.
- When reading/writing multiple bytes of data in memory, which order are the bytes put in?
 - Little Endian: byte at “xxxxx000₂” is least significant
 - [7 6 5 4 3 2 1 0]
 - Big Endian: byte at “xxxxx000₂” is most significant
 - [0 1 2 3 4 5 6 7]
- x86 => little endian
- SPARC/PowerPC => big endian
- MIPS/ARM/IA64/PA-RISC => mode bit

Assuming a Little Endian architecture. If the 4-byte value 0x12345678 is stored at address 1000₂ the value of the byte stored at address 1011₂ is:

- A: 11
- B: 0x12
- C: 0x34
- D: 0x56
- E: 0x78

Memory Addressing

Each data value is stored in memory at a “memory address”.

- What does a memory address (address) specify?
 - Byte-addressable (desktop, notebook)
 - Address specifies a multiple of 8
 - Access power of 2 number of bytes
 - Word addressable (often found in embedded processors)
 - Word size = number of bits used in arithmetic operations such as addition (8, 16, 24-bits common word sizes for embedded processors)
 - Address specifies which “word” to access in memory.
- When reading/writing multiple bytes of data in memory, which order are the bytes put in?
 - Little Endian: byte at “xxxxx000₂” is least significant
 - [7 6 5 4 3 2 1 0]
 - Big Endian: byte at “xxxxx000₂” is most significant
 - [0 1 2 3 4 5 6 7]
- x86 => little endian
- SPARC/PowerPC => big endian
- MIPS/ARM/IA64/PA-RISC => mode bit

Assuming a Little Endian architecture. If the 4-byte value 0x12345678 is stored at address 1000₂ the value of the byte stored at address 1011₂ is:

A: 11

B: 0x12 ✓

C: 0x34

D: 0x56

E: 0x78

Memory Alignment

- In many architectures (e.g., RISC), addresses must be “aligned”;
i.e., (byte address) modulo (length in bytes) = 0
- For others (x86) there may be performance benefits if accesses are aligned

Value of 3 low-order bits of byte address								
width of object	0	1	2	3	4	5	6	7
1 byte								
2 bytes (half word)								
2 bytes (half word)								
4 bytes (word)								
4 bytes (word)								
4 bytes (word)								
4 bytes (word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								

Memory Alignment

- In many architectures (e.g., RISC), addresses must be “aligned”;
i.e., (byte address) modulo (length in bytes) = 0
- For others (x86) there may be performance benefits if accesses are aligned

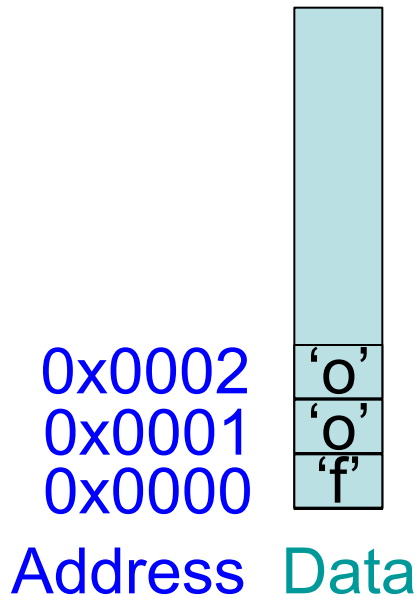
Bad: Two
memory
accesses
required

	Value of 3 low-order bits of byte address							
width of object	0	1	2	3	4	5	6	7
1 byte								
2 bytes (half word)								
2 bytes (half word)								
4 bytes (word)								
4 bytes (word)								
4 bytes (word)								
4 bytes (word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								
8 bytes (double word)								

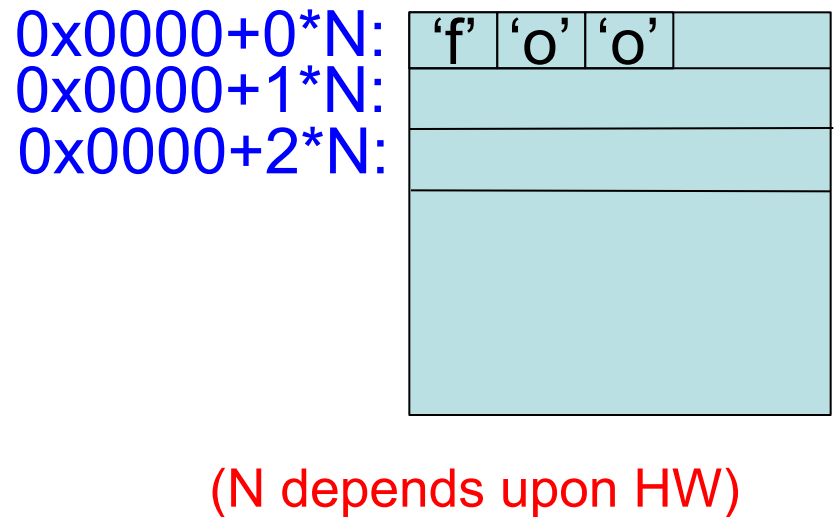
Why Does Alignment Matter?

Consider Memory Hardware...

Programmer's View:

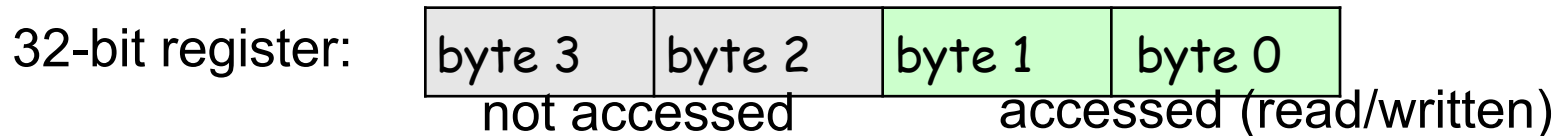


Hardware:



Partial Register Access

A partial register access reads or writes only a portion of a full register (e.g., update only least low order byte of 32-bit register).



Example: Partial write of 2-byte value 0xABCD to 4-byte (32-bit) register containing 0x12345678 results in 0x1234ABCD

Instruction sets that allow partial register **write** complicate hardware support for high performance implementations (the reasons why will be more clear later in the course when we talk about pipelining and tomasulo's algorithm).

Partial register **write** found on x86 (8-bit => 16-bit => 32-bit)
Also on historical ISAs such as VAX and IBM 360

Partial register write not found in MIPS64 or other RISC ISAs

Addressing Modes

- Recall Notation:
 - “Regs” is array of N-bit registers
 - (for MIPS64, 32x64-bit registers)
 - Regs[x] is the 64-bit quantity in register “x”
 - “Mem” is array of bytes
 - Mem[x] is “m” bytes starting at address “x”
 - “m” will be implied by usage
- Terminology:
 - “Effective Address” = actual memory address calculated by instruction (just before accessing memory).

Addressing Modes

	<u>Mode</u>	<u>Example</u>	<u>Meaning</u>
1.	Register	Add R4, <u>R3</u>	Regs[R4] <- Regs[R4] + <u>Regs[R3]</u>
2.	Immediate	Add R4, <u>#3</u>	Regs[R4] <- Regs[R4] + <u>3</u>
3.	Displacement	Add R4, <u>100(R1)</u>	Regs[R4] <- Regs[R4] + <u>Mem[100 + Regs[R1]]</u>
4.	Register Indirect	Add R4, <u>(R1)</u>	Regs[R4] <- Regs[R4] + <u>Mem[Reg[R1]]</u>
5.	Indexed	Add R3, <u>(R1+R2)</u>	Regs[R3] <- Regs[R3] + <u>Mem[Regs[R1]+Regs[R2]]</u>

Addressing Modes

<u>Mode</u>		
6.	Direct	Add R1, <u>(1001)</u> $\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[1001]$
7.	Memory Indirect	Add R1, <u>@(R3)</u> $\text{Regs}[R1] \leftarrow \text{Regs}[R1] +$ $\text{Mem}[\text{Mem}[\text{Regs}[R3]]]$
8.	Autoincrement	Add R1, <u>(R2)+</u> $\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$ $\text{Regs}[R2] \leftarrow \text{Regs}[R2] + d$
9.	Autodecrement	Add R1, <u>-(R2)</u> $\text{Regs}[R2] \leftarrow \text{Regs}[R2] - d$ $\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$
10.	Scaled	Add R1, <u>100(R2)[R3]</u> $\text{Regs}[R1] \leftarrow \text{Regs}[R1] +$ $\text{Mem}[100 + \text{Regs}[R2] + \text{Regs}[R3] * d]$

Addressing Modes

Mode

- | | | | |
|----|-----------------|-----------------------|---|
| 6. | Direct | Add R1, <u>(1001)</u> | Regs[R1] <- Regs[R1] + <u>Mem[1001]</u> |
| 7. | Memory Indirect | Add R1, <u>@(R3)</u> | Regs[R1] <- Regs[R1] +
<u>Mem[Mem[Regs[R3]]]</u> |
| 8. | Autoincrement | Add R1, <u>(R2)+</u> | Regs[R1] <- Regs[R1] + <u>Mem[Regs[R2]]</u>
<u>Regs[R2] <- Regs[R2] + d</u> |

Regs[R2] <- Regs[R2] - d
Regs[R1] + Mem[Regs[R2]]
 Regs[R1] <- Regs[R1] +
+ Regs[R2] + Regs[R3] * d]

Is it possible to implement a program that uses instructions with “memory indirect” addressing by replacing those instructions with instructions that do not use “memory indirect” addressing?

- A: Yes
 B: No
 C: Not sure

Addressing Modes

	Mode		
6.	Direct	Add R1, <u>(1001)</u>	Regs[R1] <- Regs[R1] + <u>Mem[1001]</u>
7.	Memory Indirect	Add R1, <u>@(R3)</u>	Regs[R1] <- Regs[R1] + <u>Mem[Mem[Regs[R3]]]</u>
8.	Autoincrement	Add R1, <u>(R2)+</u>	Regs[R1] <- Regs[R1] + <u>Mem[Regs[R2]]</u> <u>Regs[R2] <- Regs[R2] + d</u>

Is it possible to implement a program that uses instructions with “memory indirect” addressing by replacing those instructions with instructions that do not use “memory indirect” addressing?

A: Yes ✓

ADD R1, @(R3) written as LD R2, 0(R3)
 ADD R1, 0(R2)

(does require using an additional register, R2)

B: No

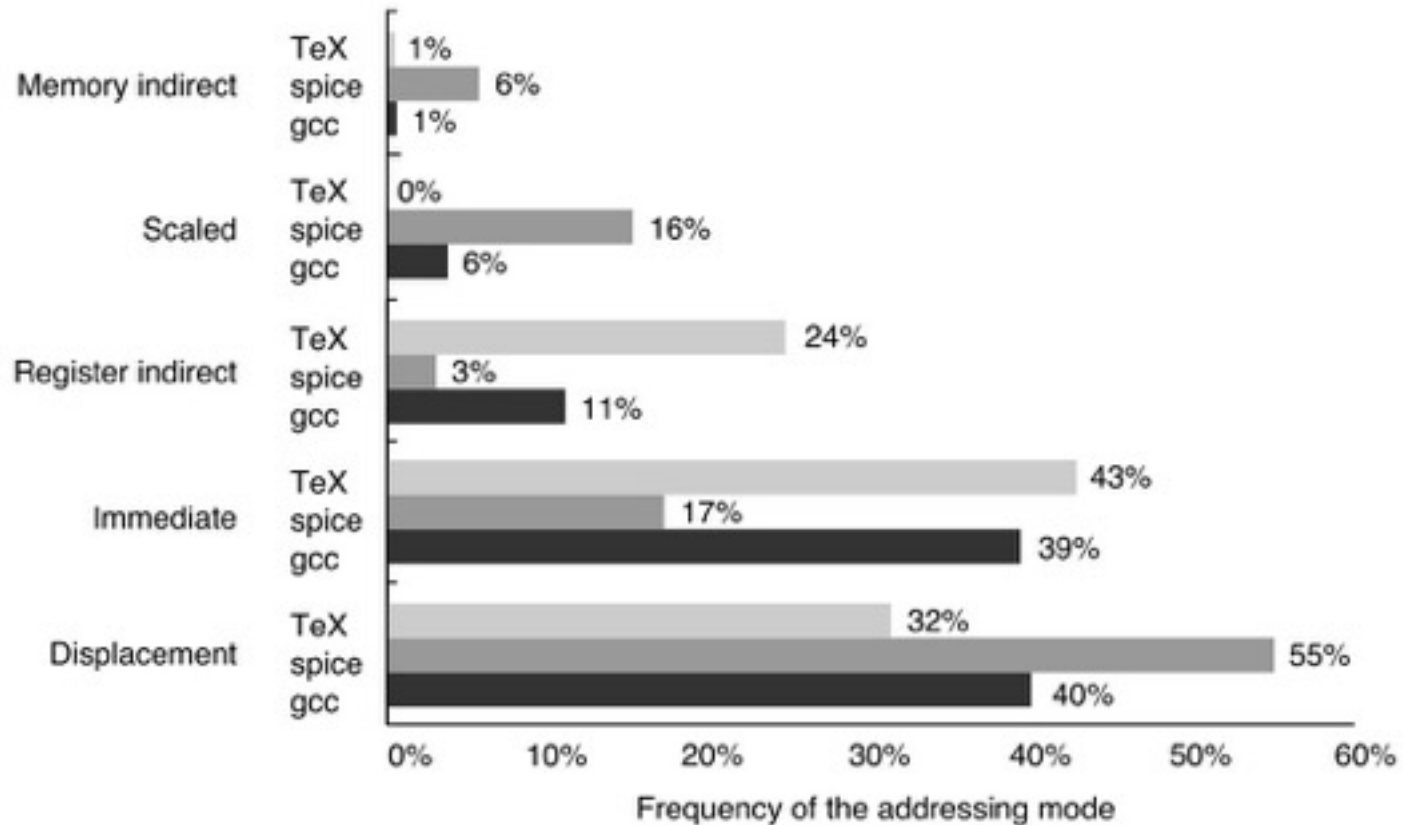
C: Not sure

Regs[R2] <- Regs[R2] - d
Regs[R1] + Mem[Regs[R2]]
Regs[R1] <- Regs[R1] +
+ Regs[R2] + Regs[R3] * d]

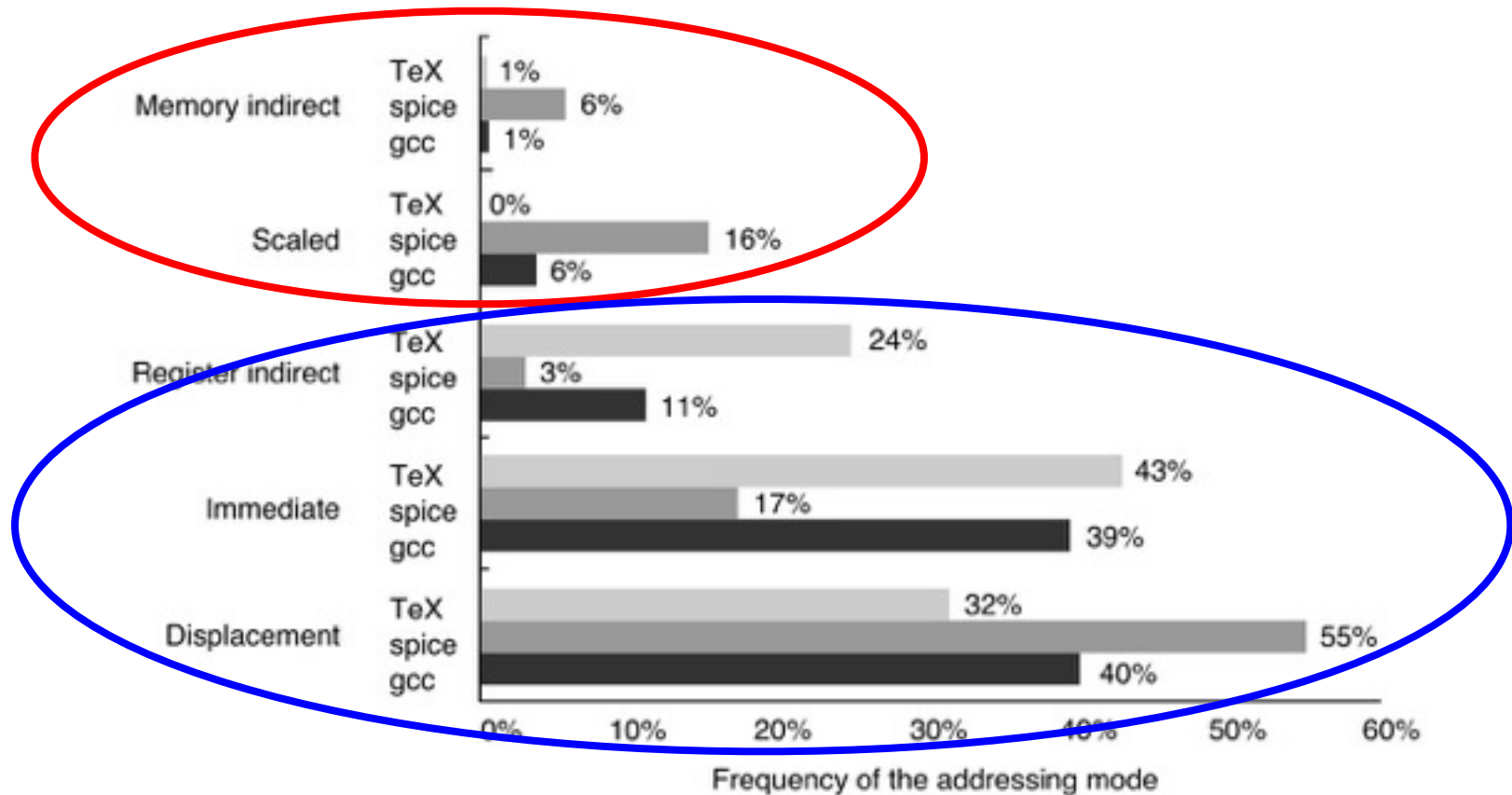
Addressing Modes

<u>Mode</u>		
6.	Direct	Add R1, <u>(1001)</u> $\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[1001]$
7.	Memory Indirect	Add R1, <u>@(R3)</u> $\text{Regs}[R1] \leftarrow \text{Regs}[R1] +$ $\text{Mem}[\text{Mem}[\text{Regs}[R3]]]$
8.	Autoincrement	Add R1, <u>(R2)+</u> $\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$ $\text{Regs}[R2] \leftarrow \text{Regs}[R2] + d$
9.	Autodecrement	Add R1, <u>-(R2)</u> $\text{Regs}[R2] \leftarrow \text{Regs}[R2] - d$ $\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$
10.	Scaled	Add R1, <u>100(R2)[R3]</u> $\text{Regs}[R1] \leftarrow \text{Regs}[R1] +$ $\text{Mem}[100 + \text{Regs}[R2] + \text{Regs}[R3] * d]$

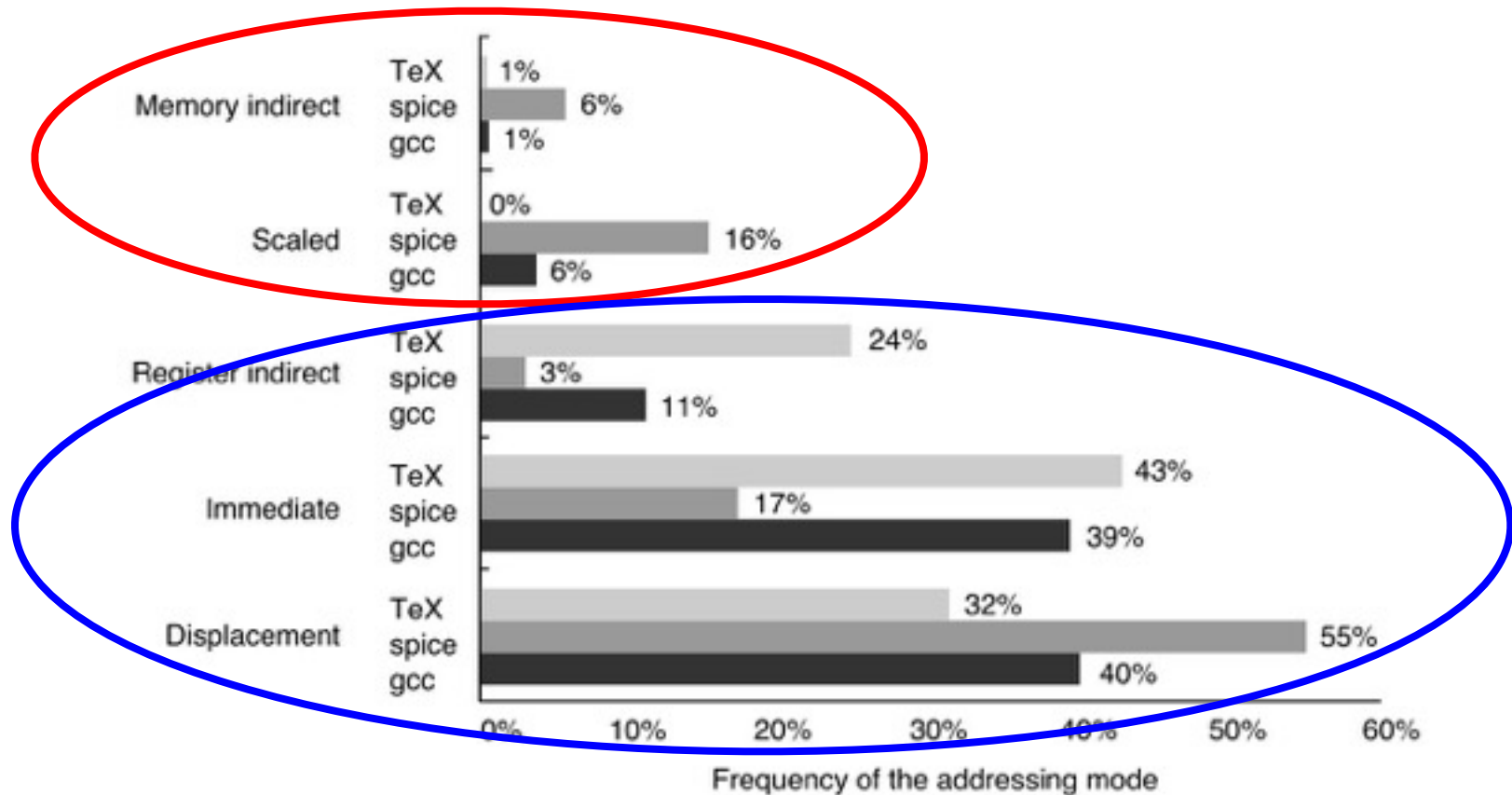
Addressing Modes



Addressing Modes

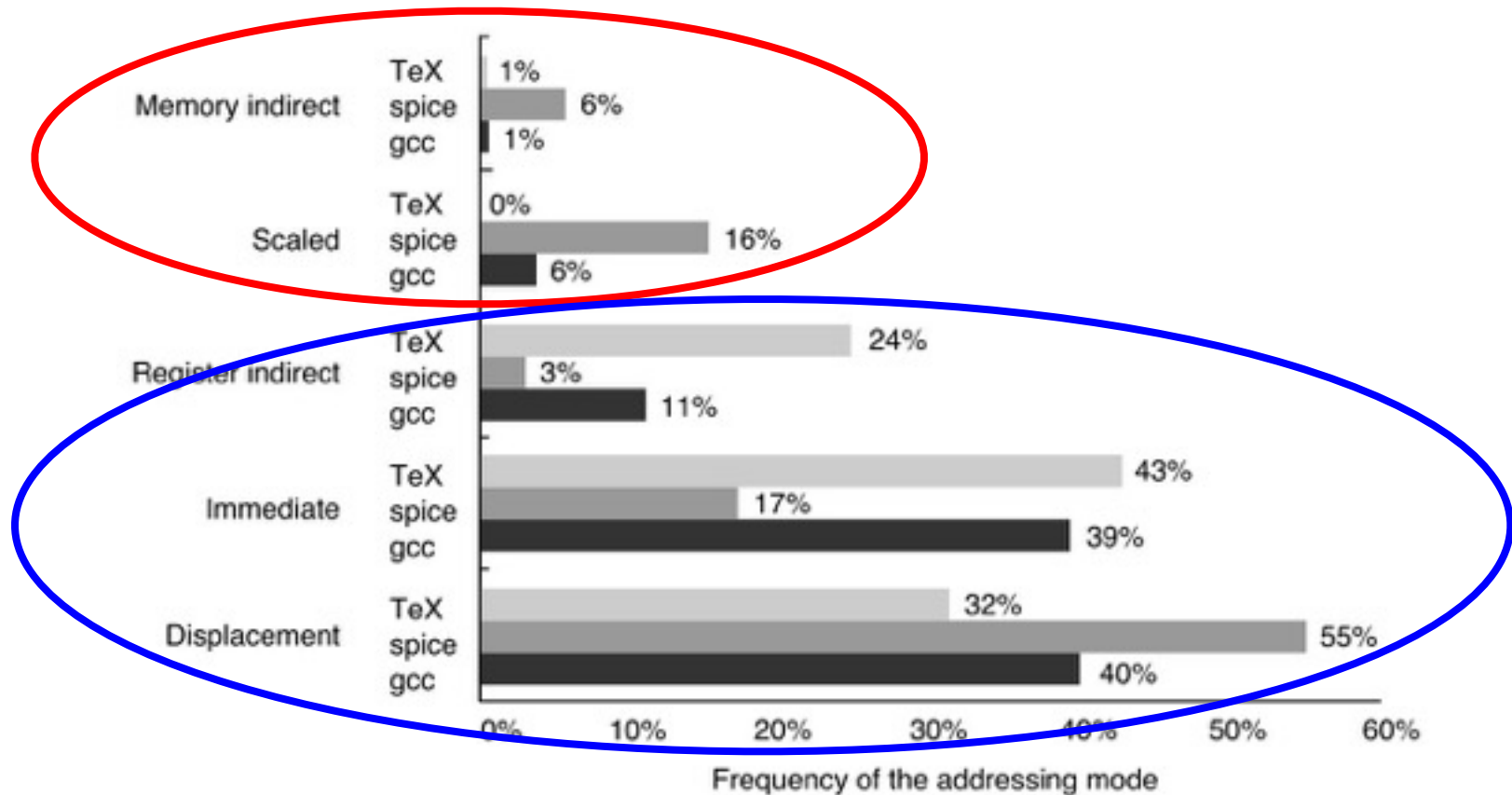


Addressing Modes



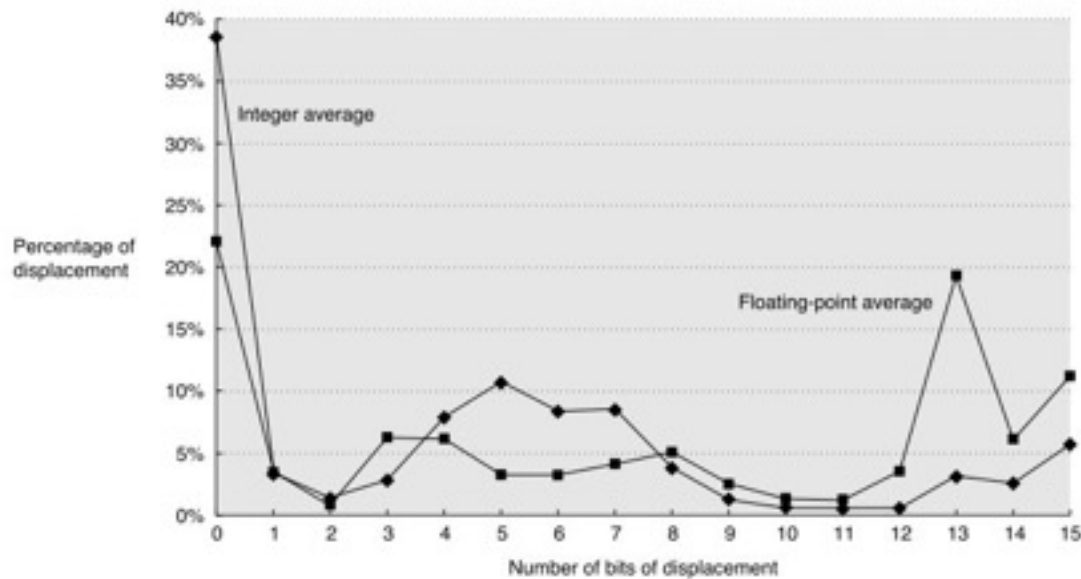
Lesson: 'fancy' addressing modes not used much by compiler

Addressing Modes



Lesson: 'fancy' addressing modes not used much by compiler
simple addressing modes used most often

Displacement Mode



- How many bits should be used to encode displacement?
- What are the tradeoffs?

Example: How Many Bits?

Imagine a RISC instruction set that supports a 24-bit displacement field and that initially $\text{Regs}[R1]=0x40$. We have a program with the following instruction:

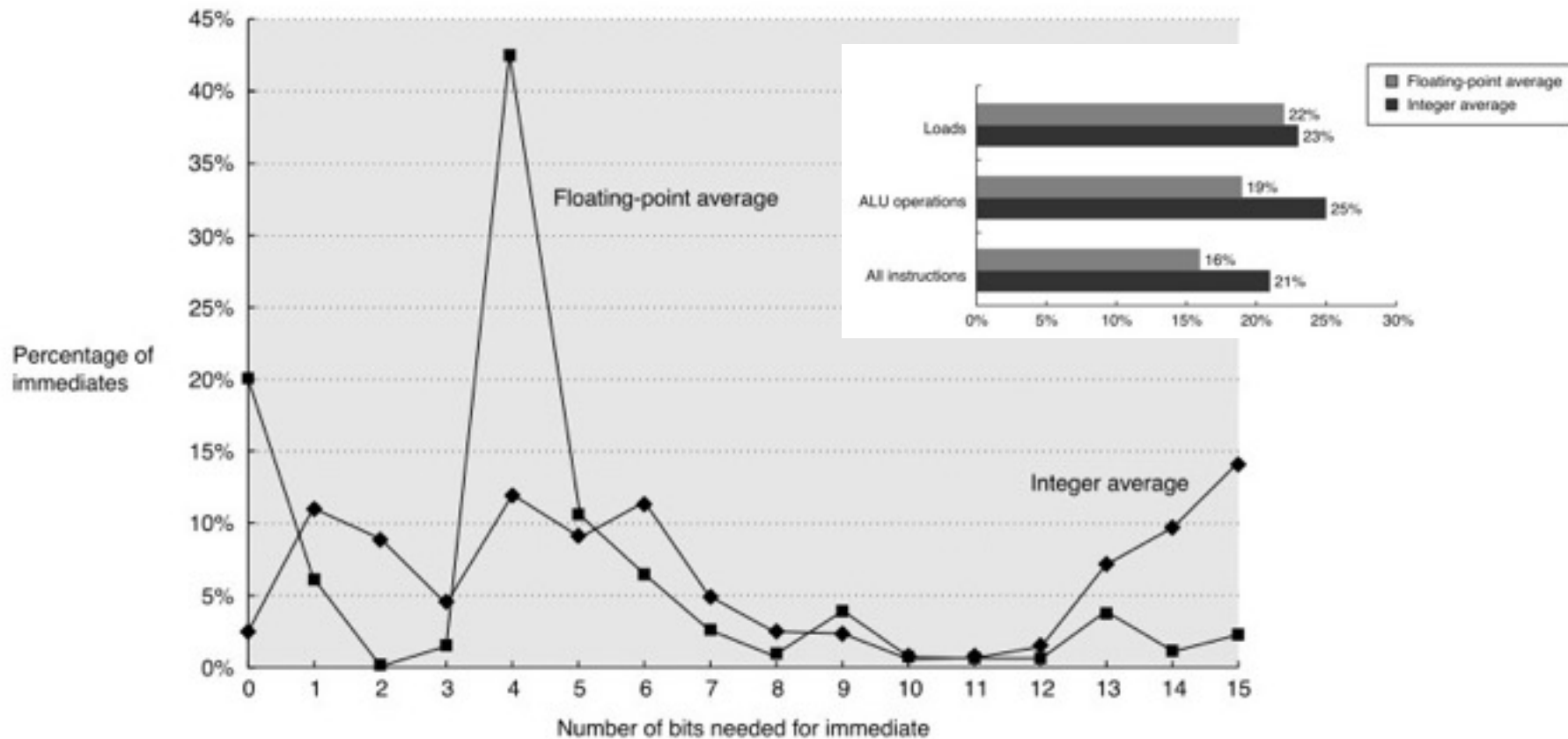
`LD R2, 0x10002(R1) ; Eff. Addr. = $0x10002+0x40=0x10042$`

NOTE: The value $0x10002$ requires at least 17 bits to represent.

We can implement the same operation using a 16-bit displacement by using multiple instructions. For example, using MIPS64 (which has a 16-bit displacement field):

`LUI R3,#1 ; $\text{Regs}[R3] = 0x10000$
DADD R4,R3,R1 ; $\text{Regs}[R4] = 0x10000+0x40=0x10040$
LD R2,2(R4) ; $\text{Eff. Addr} = 0x10040+0x2=0x10042$`

Usage of Immediate Operand



- What are implications for ISA design?

Question

Assuming any size displacement is allowed, a program contains 25% loads with 50% of these loads having displacement of 8 bits or less and all loads have displacement of 16 bits or less.

1. If we require length of load to be 16 + displacement size bits, **which uses less instruction memory**: using (a) **8-bit** or (b) 16-bit displacements for loads (assuming all other instructions are 16-bits)? If a displacement exceeds the size of the displacement field an additional 16 + displacement size instruction is required.
2. What if all instructions must have the same “width” in bits? (i.e., all instructions are (a) all 24-bits or (b) all 32-bits)

For part 1(a) below, on average, how many bits are required if we need to use 8bits to represent the “displacement”?

- A: 16
- B: 21
- C: 24
- D: 32
- E: Not sure

Assuming any size displacement is allowed, a program contains 25% loads with 50% of these loads having displacement of 8 bits or less and all loads have displacement of 16 bits or less.

1. If we require length of load to be 16 + displacement size bits, **which uses less instruction memory**: using (a) **8-bit** or (b) 16-bit displacements for loads (assuming all other instructions are 16-bits)? If a displacement exceeds the size of the displacement field an additional 16 + displacement size instruction is required.
2. What if all instructions must have the same “width” in bits? (i.e., all instructions are (a) all 24-bits or (b) all 32-bits)

Question

Assuming any size displacement is allowed, a program contains 25% loads with 50% of these loads having displacement of 8 bits or less and all loads have displacement of 16 bits or less.

1. If we require length of load to be 16 + displacement size bits, **which uses less instruction memory**: using (a) **8-bit** or (b) 16-bit displacements for loads (assuming all other instructions are 16-bits)? If a displacement exceeds the size of the displacement field an additional 16 + displacement size instruction is required.
2. What if all instructions must have the same “width” in bits? (i.e., all instructions are (a) all 24-bits or (b) all 32-bits)

For part 1 below, on average, how many bits are required if we need to use 8bits to represent the “displacement”?

A: 16

B: 21 ✓

C: 24

D: 32

E: Not sure

Assuming any size displacement is allowed, a program contains 25% loads with 50% of these loads having displacement of 8 bits or less and all loads have displacement of 16 bits or less.

1. If we require length of load to be 16 + displacement size bits, **which uses less instruction memory**: using (a) **8-bit** or (b) 16-bit displacements for loads (assuming all other instructions are 16-bits)? If a displacement exceeds the size of the displacement field an additional 16 + displacement size instruction is required.
2. What if all instructions must have the same “width” in bits? (i.e., all instructions are (a) all 24-bits or (b) all 32-bits)

Question

Assuming any size displacement is allowed, a program contains 25% loads with 50% of these loads having displacement of 8 bits or less and all loads have displacement of 16 bits or less.

1. If we require length of load to be 16 + displacement size bits, **which uses less instruction memory**: using (a) **8-bit** or (b) 16-bit displacements for loads (assuming all other instructions are 16-bits)? If a displacement exceeds the size of the displacement field an additional 16 + displacement size instruction is required.
2. What if all instructions must have the same “width” in bits? (i.e., all instructions are (a) all 24-bits or (b) all 32-bits)

Answer

Answer

Part 1:

Answer

Part 1:

(a) 8-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+8) + 0.25 \cdot 0.5 \cdot (16+8)$

Answer

Part 1:

(a) 8-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+8) + 0.25 \cdot 0.5 \cdot (16+8)$
 $= 21$ bits per (original) operation

Answer

Part 1:

(a) 8-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+8) + 0.25 \cdot 0.5 \cdot (16+8)$
 $= 21$ bits per (original) operation

(b) 16-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+16)$

Answer

Part 1:

(a) 8-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+8) + 0.25 \cdot 0.5 \cdot (16+8)$
 $= 21$ bits per (original) operation

(b) 16-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+16)$
 $= 20$ bits per (original) operation

Answer

Part 1:

(a) 8-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+8) + 0.25 \cdot 0.5 \cdot (16+8)$
 $= 21$ bits per (original) operation

(b) 16-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+16)$
 $= 20$ bits per (original) operation

Part 2:

Answer

Part 1:

(a) 8-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+8) + 0.25 \cdot 0.5 \cdot (16+8)$
 $= 21$ bits per (original) operation

(b) 16-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+16)$
 $= 20$ bits per (original) operation

Part 2:

(a) 8-bit $\Rightarrow 0.75 \cdot 24 + 0.25 \cdot 24 + 0.25 \cdot 0.5 \cdot 24$

Answer

Part 1:

(a) 8-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+8) + 0.25 \cdot 0.5 \cdot (16+8)$
 $= 21$ bits per (original) operation

(b) 16-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+16)$
 $= 20$ bits per (original) operation

Part 2:

(a) 8-bit $\Rightarrow 0.75 \cdot 24 + 0.25 \cdot 24 + 0.25 \cdot 0.5 \cdot 24$
 $= 27$ bits per (original) operation

Answer

Part 1:

(a) 8-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+8) + 0.25 \cdot 0.5 \cdot (16+8)$
 $= 21$ bits per (original) operation

(b) 16-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+16)$
 $= 20$ bits per (original) operation

Part 2:

(a) 8-bit $\Rightarrow 0.75 \cdot 24 + 0.25 \cdot 24 + 0.25 \cdot 0.5 \cdot 24$
 $= 27$ bits per (original) operation

(b) 16-bit $\Rightarrow 0.75 \cdot 32 + 0.25 \cdot 32$

Answer

Part 1:

(a) 8-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+8) + 0.25 \cdot 0.5 \cdot (16+8)$
 $= 21$ bits per (original) operation

(b) 16-bit $\Rightarrow 0.75 \cdot 16 + 0.25 \cdot (16+16)$
 $= 20$ bits per (original) operation

Part 2:

(a) 8-bit $\Rightarrow 0.75 \cdot 24 + 0.25 \cdot 24 + 0.25 \cdot 0.5 \cdot 24$
 $= 27$ bits per (original) operation

(b) 16-bit $\Rightarrow 0.75 \cdot 32 + 0.25 \cdot 32$
 $= 32$ bits per (original) operation

Type and Size of Operands

- Character (8-bit, 16-bit)
- Signed Integer (64-bit, 32-bit, 16-bit, 8-bit)
 - (2's complement)
- Unsigned (64-bit, 32-bit, 16-bit, 8-bit)
- Single precision floating point (32-bit)
- Double precision floating point (64-bit)
- BCD (binary coded decimal)
- Vertex (4x 32bit floating point)
- Color (RGBA)
- Fixed-point

Typical ISA Operations

There are several basic types of operations that are found in most instruction sets. These are listed below.

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic read-modify-write)

Top 10 IA-32 Instructions

Rank	Instruction	Integer Average (Percent total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
<hr/>		
	Total	96%

- Simple instructions dominate instruction frequency

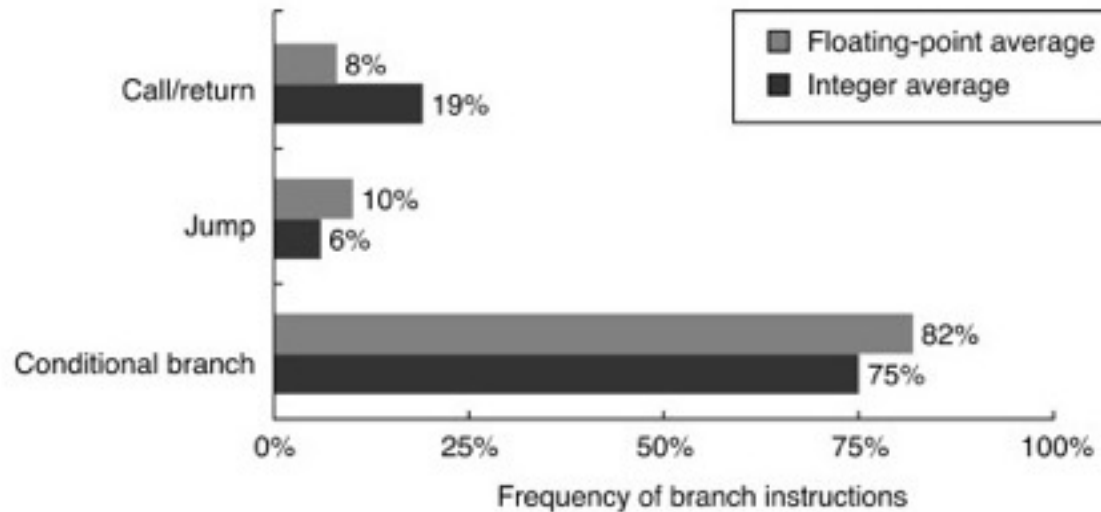
Instructions for Control Flow

- Four basic types
 - Conditional Branches
 - Jumps
 - Procedure Calls
 - Procedure Returns

terminology (H&P, MIPS64):

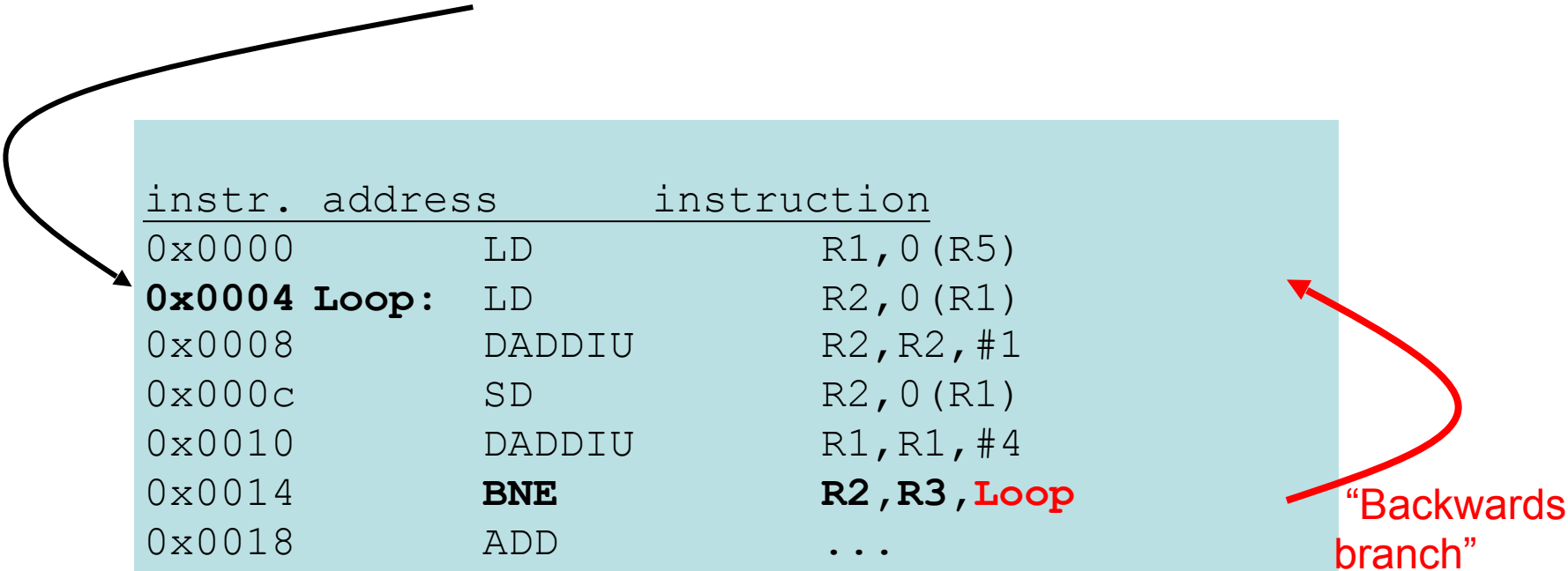
jump = change in control flow unconditional.

branch = conditional change in control flow.



Addressing Modes for Control Flow Instructions

- Destination (“target”) address:



instr.	address	instruction	
0x0000		LD	R1, 0(R5)
0x0004	Loop:	LD	R2, 0(R1)
0x0008		DADDIU	R2, R2, #1
0x000c		SD	R2, 0(R1)
0x0010		DADDIU	R1, R1, #4
0x0014		BNE	R2, R3, Loop
0x0018		ADD	...

“Backwards
branch”

Addressing Control Flow

- Destination (“target”)

Suppose we want instructions to be as small as possible (e.g., use as few bits as possible). What value should we use for “Loop” in the BNE instruction?

- A: 0x4
- B: 0x14
- C: 0xC
- D: -0x10
- E: 0xFB

instr.	address	instruction	
0x0000		LD	R1, 0 (R5)
0x0004	Loop:	LD	R2, 0 (R1)
0x0008		DADDIU	R2, R2, #1
0x000c		SD	R2, 0 (R1)
0x0010		DADDIU	R1, R1, #4
0x0014		BNE	R2, R3, Loop
0x0018		ADD	...

“Backwards branch”

Addressing Control Flow

- Destination (“target”)

Suppose we want instructions to be as small as possible (e.g., use as few bits as possible). What value should we use for “Loop” in the BNE instruction?

A: 0x4

B: 0x14

C: 0xC

D: -0x10

E: 0xFB ✓ ??? = $(0x4 - (0x14 + 0x4)) / 4$

instr.	address	instruction
0x0000	LD	R1, 0(R5)
0x0004 Loop:	LD	R2, 0(R1)
0x0008	DADDIU	R2, R2, #1
0x000c	SD	R2, 0(R1)
0x0010	DADDIU	R1, R1, #4
0x0014	BNE	R2, R3, Loop
0x0018	ADD	...

“Backwards branch”

PC-relative Addressing

- Recall that the “program counter” (PC) provides the memory address of the instruction to execute.
- For PC-relative addressing, we compute the new program counter address by using the address of the branch instruction (stored in the PC), and adding an offset to this address:

$\text{destination} := (\text{PC of branch}) + \text{offset}$

- Benefits of PC-relative addressing:
 1. offset encoded using less bits than PC... why?
 2. “position independence” (helps “linking”, esp. in DLLs)

PC-relative A

- Recall that the “program counter” is the address of the instruction to execute.
- For PC-relative addressing, we calculate the destination address by using the address of the instruction (stored in the PC), and adding a

destination := (PC of branch instruction + offset)

- Benefits of PC-relative addressing:
 1. offset encoded using less bits than PC... why?
 2. “position independence” (helps “linking”, esp. in DLLs)

Why is it that we can encode offset using less bits if we use PC-relative addressing?

A: Because least significant bits are always zero since instructions are aligned in memory.

B: Because instructions are always located starting at an offset from address zero (e.g., 0x400000).

C: Because the distance between a branch instruction and the target of the branch tends to be small in real programs.

D: None of the above.

E: Not sure.

PC-relative A

- Recall that the “program counter” is the address of the instruction to execute.
- For PC-relative addressing, we calculate the destination address by using the address of the instruction (stored in the PC), and adding a

destination := (PC of branch instruction + offset)

- Benefits of PC-relative addressing:
 1. offset encoded using less bits than PC... why?
 2. “position independence” (helps “linking”, esp. in DLLs)

Why is it that we can encode offset using less bits if we use PC-relative addressing?

A: Because least significant bits are always zero since instructions are aligned in memory.

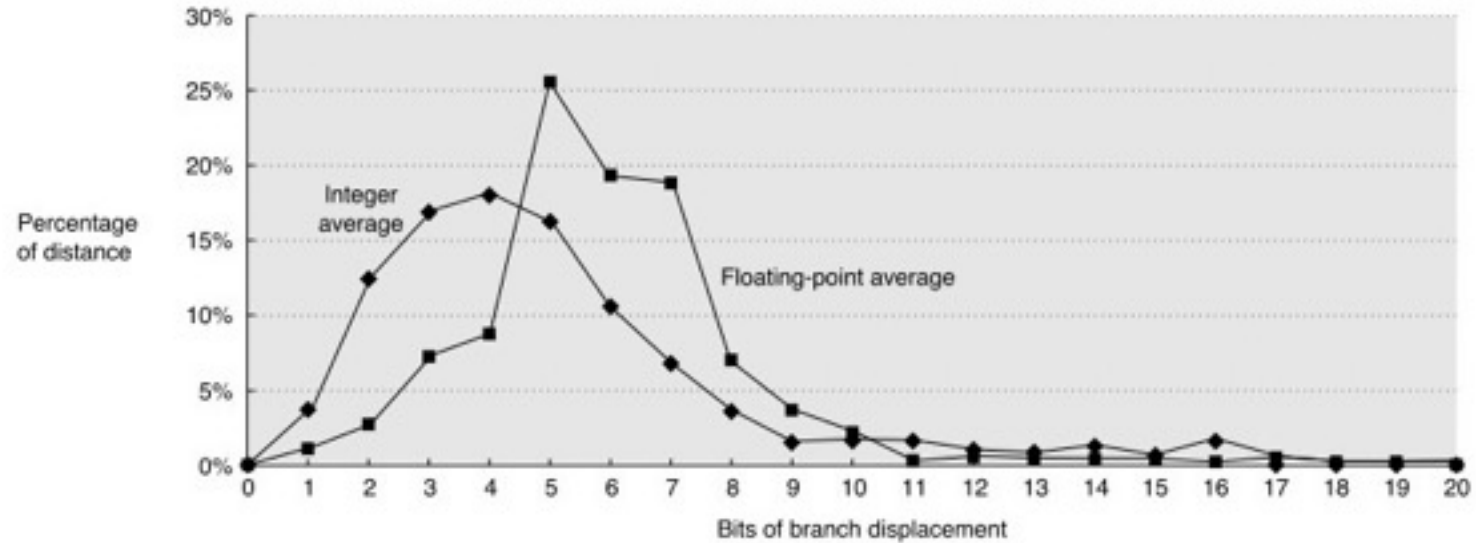
B: Because instructions are always located starting at an offset from address zero (e.g., 0x400000).

C: Because distance between branch instruction and target of the branch tends to be small in real programs. ✓

D: None of the above.

E: Not sure.

How many bits?



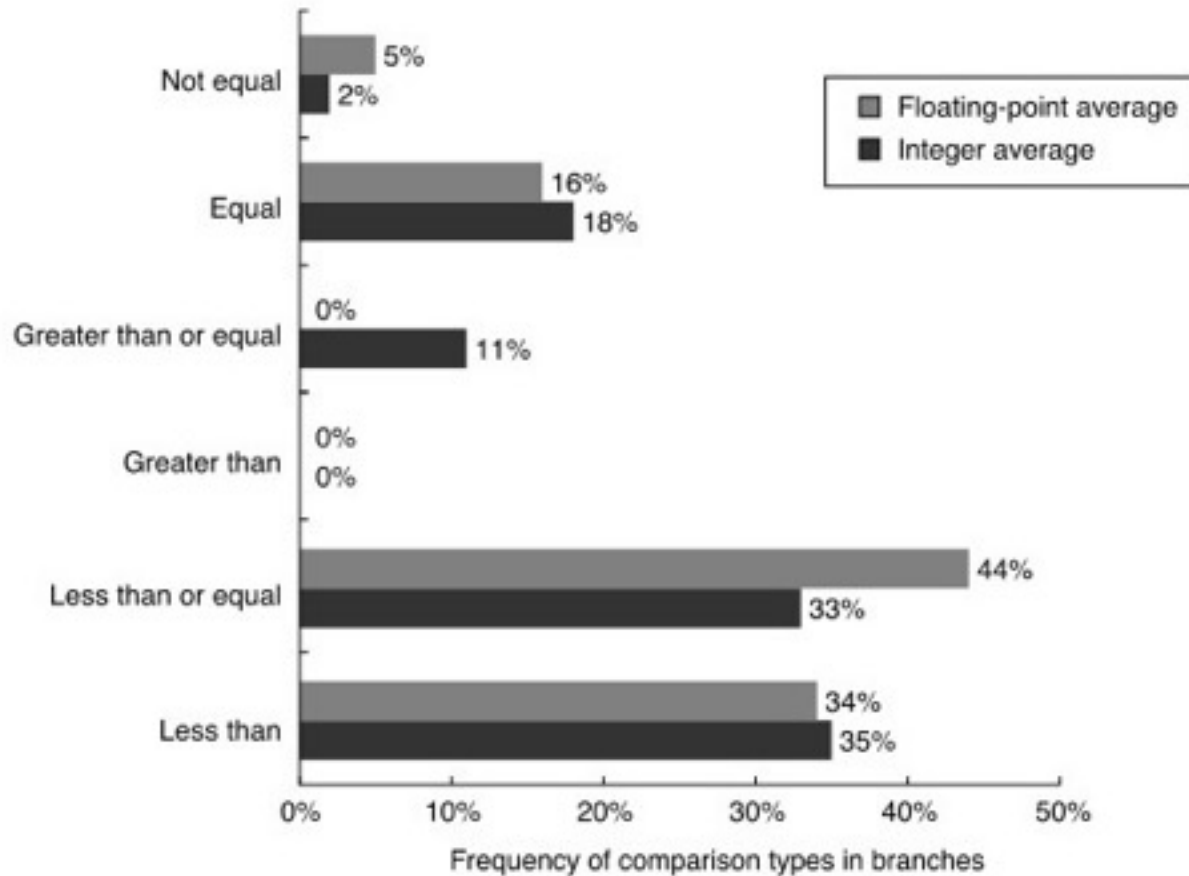
- 8 bits often enough!

Destination Register

- PC-relative addressing not helpful if target is not known at compile time.
 - procedure return
 - indirect jumps
- Use a register to specify destination PC
- Indirect jumps useful for:
 - switch statements
 - virtual functions (C++, Java)
 - function pointers (C++)
 - dynamically shared libraries

Conditional Branch Options

- Most branches depend upon simple comparisons:



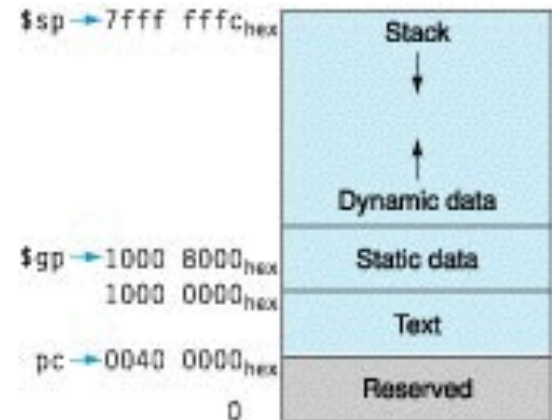
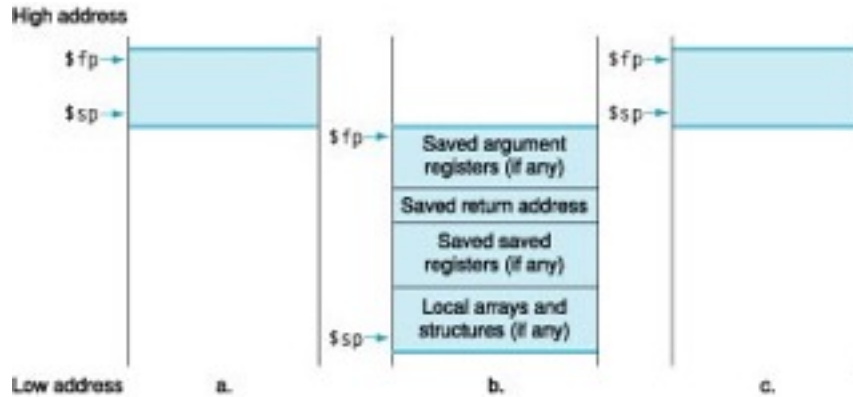
Conditional Branch Options

How do we know if a branch should update the PC to the target of the branch? Here are some approaches used in real machines:

- Condition codes: Read a special “flag” bit to determine whether branch is taken. Flag bit set by an earlier instruction.
 - Branch condition computed as side effect of earlier instructions
 - Benefits: Can reduce instructions
 - Drawbacks: Constrains ordering of instructions, can complicate hardware implementation
- Condition register: E.g., branch if contents of register non-zero
 - Test arbitrary register
 - Benefit: better for compiler; easier hardware implementation
 - Drawbacks: increases instruction count
- Compare and Branch: Combine comparison with branch
 - Benefit: Saves instructions
 - Drawback: May increase cycle time

Procedure Invocation

While we are talking about control flow instructions, let's consider how function calls and returns work:



- Key issue: variables local to a function should not be modified by function calls. Such variables are often stored in registers to improve performance.
- Who saves registers on a function call?
 - Caller save: Before a “call instruction”, save registers
 - Callee save: First thing done when starting a function is to save registers.
- One approach sometimes better than other (depends upon program).
- In practice: Set agreed upon standard (application binary interface)
- Alternative: Hardware does saving/restoring during call instruction (inefficient).

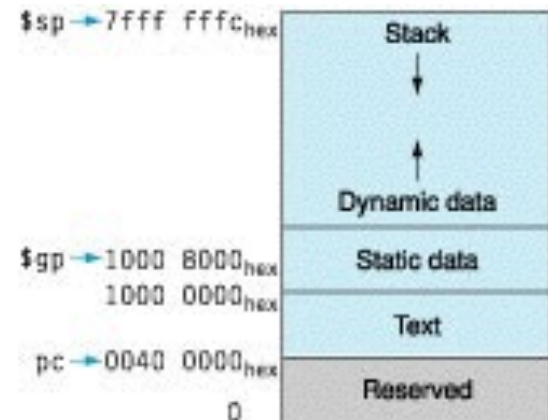
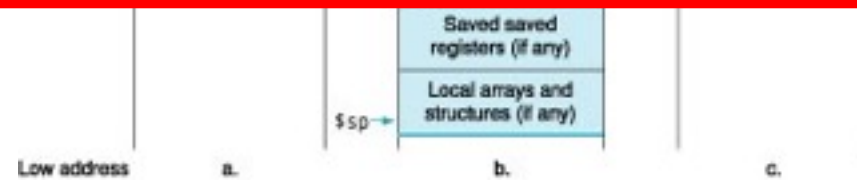
Does this load instruction modify R16?

LW R16,0(R1)

- A: Yes, 100% sure
- B: Yes, but not 100% sure
- C: Not sure either way
- D: No, but not 100% sure
- E: No, 100% sure

Invocation

Now instructions, let's
see how they work:



- Key issue: variables local to a function should not be modified by function calls. Such variables are often stored in registers to improve performance.
- Who saves registers on a function call?
 - Caller save: Before a “call instruction”, save registers
 - Callee save: First thing done when starting a function is to save registers.
- One approach sometimes better than other (depends upon program).
- In practice: Set agreed upon standard (application binary interface)
- Alternative: Hardware does saving/restoring during call instruction (inefficient).

Does this load instruction modify R16?

LW R16,0(R1)

A: Yes, 100% sure ✓

B: Yes, but not 100% sure

C: Not sure either way

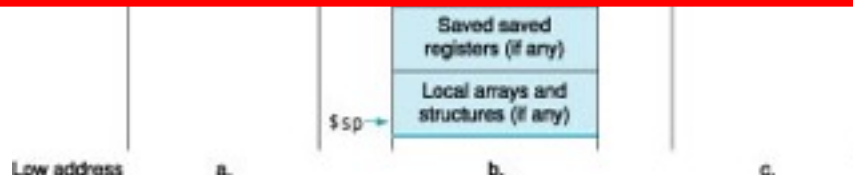
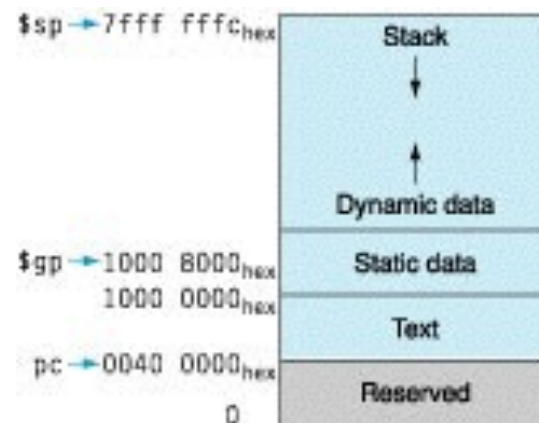
D: No, but not 100% sure

E: No, 100% sure

Invocation

Now instructions, let's

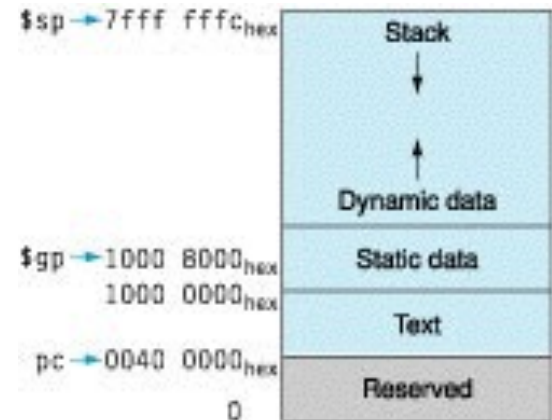
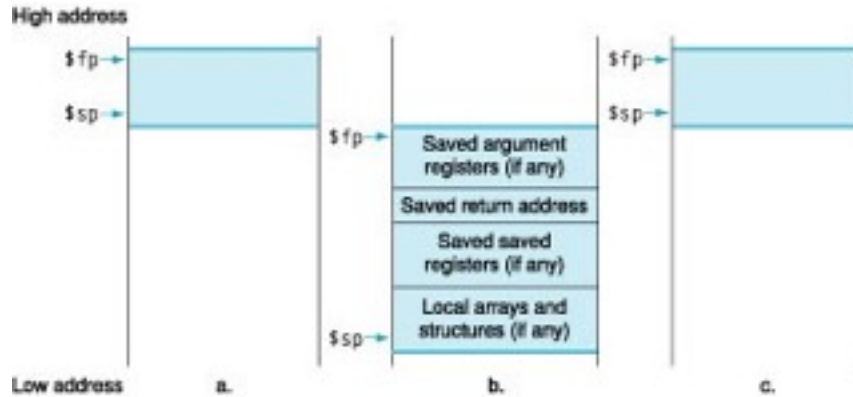
things work:



- Key issue: variables local to a function should not be modified by function calls. Such variables are often stored in registers to improve performance.
- Who saves registers on a function call?
 - Caller save: Before a “call instruction”, save registers
 - Callee save: First thing done when starting a function is to save registers.
- One approach sometimes better than other (depends upon program).
- In practice: Set agreed upon standard (application binary interface)
- Alternative: Hardware does saving/restoring during call instruction (inefficient).

Procedure Invocation

While we are talking about control flow instructions, let's consider how function calls and returns work:



- Key issue: variables local to a function should not be modified by function calls. Such variables are often stored in registers to improve performance.
- Who saves registers on a function call?
 - Caller save: Before a “call instruction”, save registers
 - Callee save: First thing done when starting a function is to save registers.
- One approach sometimes better than other (depends upon program).
- In practice: Set agreed upon standard (application binary interface)
- Alternative: Hardware does saving/restoring during call instruction (inefficient).

Procedure

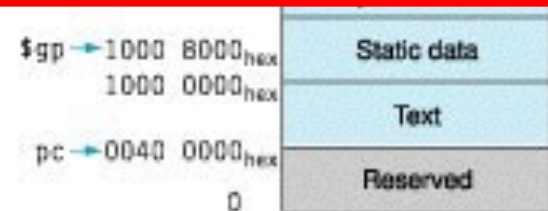
While we are talking about control flow, let's also consider how function calls and return work.



Does this store instruction modify R16?

SW R16,0(R1)

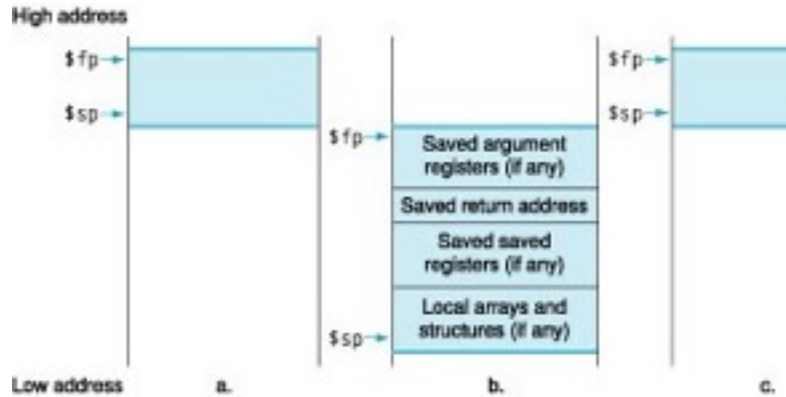
- A: Yes, 100% sure
- B: Yes, but not 100% sure
- C: Not sure either way
- D: No, but not 100% sure
- E: No, 100% sure



- Key issue: variables local to a function should not be modified by function calls. Such variables are often stored in registers to improve performance.
- Who saves registers on a function call?
 - Caller save: Before a “call instruction”, save registers
 - Callee save: First thing done when starting a function is to save registers.
- One approach sometimes better than other (depends upon program).
- In practice: Set agreed upon standard (application binary interface)
- Alternative: Hardware does saving/restoring during call instruction (inefficient).

Procedure

While we are talking about control flow, let's also consider how function calls and return work.



Does this store instruction modify R16?

SW R16,0(R1)

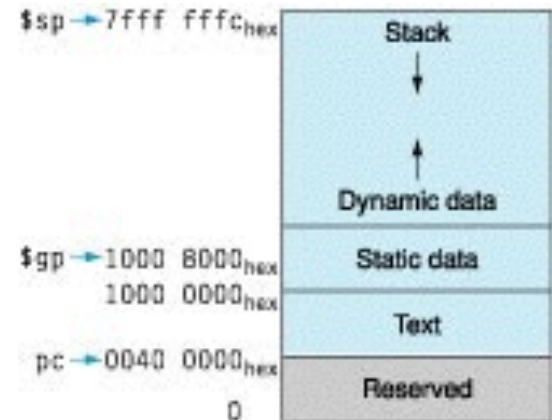
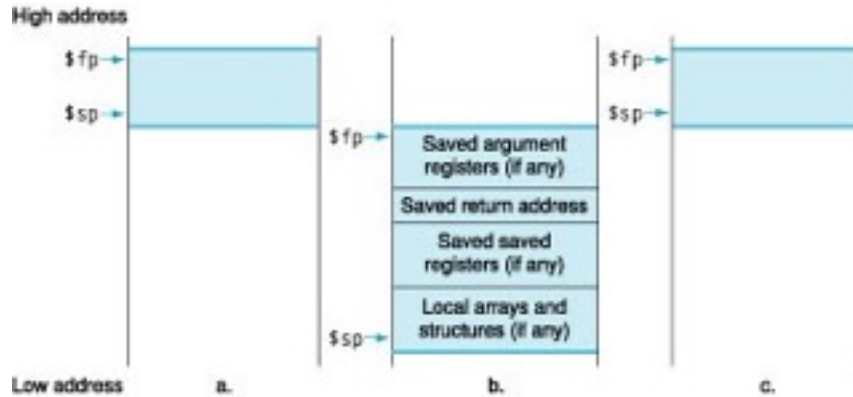
- A: Yes, 100% sure
- B: Yes, but not 100% sure
- C: Not sure either way
- D: No, but not 100% sure
- E: No, 100% sure ✓



- Key issue: variables local to a function should not be modified by function calls. Such variables are often stored in registers to improve performance.
- Who saves registers on a function call?
 - Caller save: Before a “call instruction”, save registers
 - Callee save: First thing done when starting a function is to save registers.
- One approach sometimes better than other (depends upon program).
- In practice: Set agreed upon standard (application binary interface)
- Alternative: Hardware does saving/restoring during call instruction (inefficient).

Procedure Invocation

While we are talking about control flow instructions, let's consider how function calls and returns work:



- Key issue: variables local to a function should not be modified by function calls. Such variables are often stored in registers to improve performance.
- Who saves registers on a function call?
 - Caller save: Before a “call instruction”, save registers
 - Callee save: First thing done when starting a function is to save registers.
- One approach sometimes better than other (depends upon program).
- In practice: Set agreed upon standard (application binary interface)
- Alternative: Hardware does saving/restoring during call instruction (inefficient).

P

While we are talking about functions, consider how functions are implemented in assembly.



Consider the following code:

```
unsigned int fib( unsigned int n )
{
    if( n == 0 ) return 0;
    else if( n == 1 ) return 1;
    else return fib(n-1) + fib(n-2);
}
```

Guess how many registers the MIPS assembly for this code uses.

A: 1 register

B: 2 or 3 registers

C: 4 or 5 registers

D: 6 or 7 registers

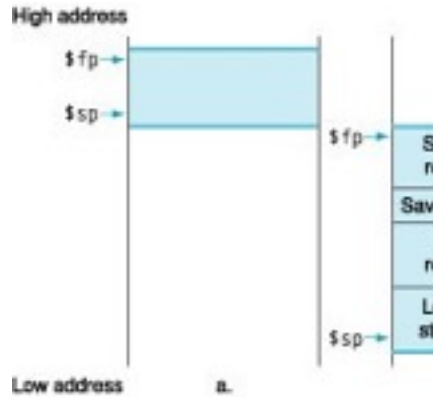
E: number of registers depends upon value of n

- Key issue: variables local to a function. Such variables are often stored in registers to improve performance.
- Who saves registers on a function call?
 - Caller save: Before a “call instruction”, save registers
 - Callee save: First thing done when starting a function is to save registers.
- One approach sometimes better than other (depends upon program).
- In practice: Set agreed upon standard (application binary interface)
- Alternative: Hardware does saving/restoring during call instruction (inefficient).

P

Answer turns out to be C (5 registers!)

While we are talking about function calls, consider how functions are implemented.



Below is what fib() looks like compiled.

```
fib:
    subu    R29,R29,24      # R29 = stack-pointer
    sw      R16,16(R29)     # R16 = temporary value
    sw      R31,20(R29)     # R31 = return address
    move    R16,R4          # R4 = function argument
    bne     R16,R0,L7       #
    move    R2,R0           # R2 = return value
    j       L11

L7:
    li      R2,0x01
    beq     R16,R2,L9
    subu    R4,R16,1
    jal     fib
    subu    R4,R16,2
    move    R16,R2
    jal     fib
    addu    R2,R16,R2
    j       L11

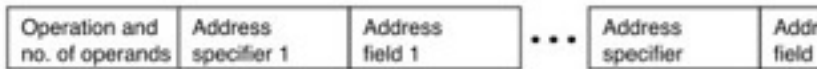
L9:
    li      R2,0x01

L11:
    lw      R31,20(R29)
    lw      R16,16(R29)
    addu    R29,R29,24
    j       R31
```

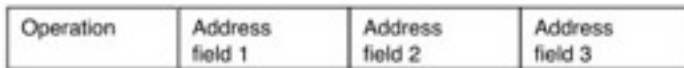
- Key issue: variables local to function. Such variables are often stored on the stack.
- Who saves registers on the stack?
 - Caller save: Before calling a function, the caller must save any registers that the function might use.
 - Callee save: First time a function is called, the callee must save any registers that it might use.
- One approach sometimes used is to have the caller save all registers that the function might use.
- In practice: Set agreed upon standard (application binary interface).
- Alternative: Hardware does saving/restoring during call instruction (inefficient).

Encoding an ISA

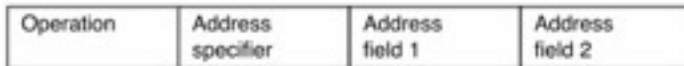
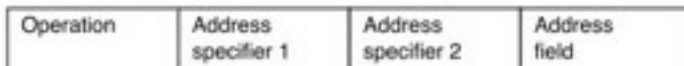
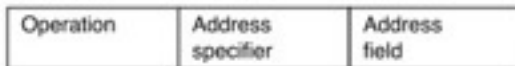
Once we decide which instructions we want, we need to specify the instruction in 1's and 0's so that digital logic gates can implement it. This is known as “encoding” the instruction set.



(a) Variable (e.g., VAX, Intel 80x86)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)



(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

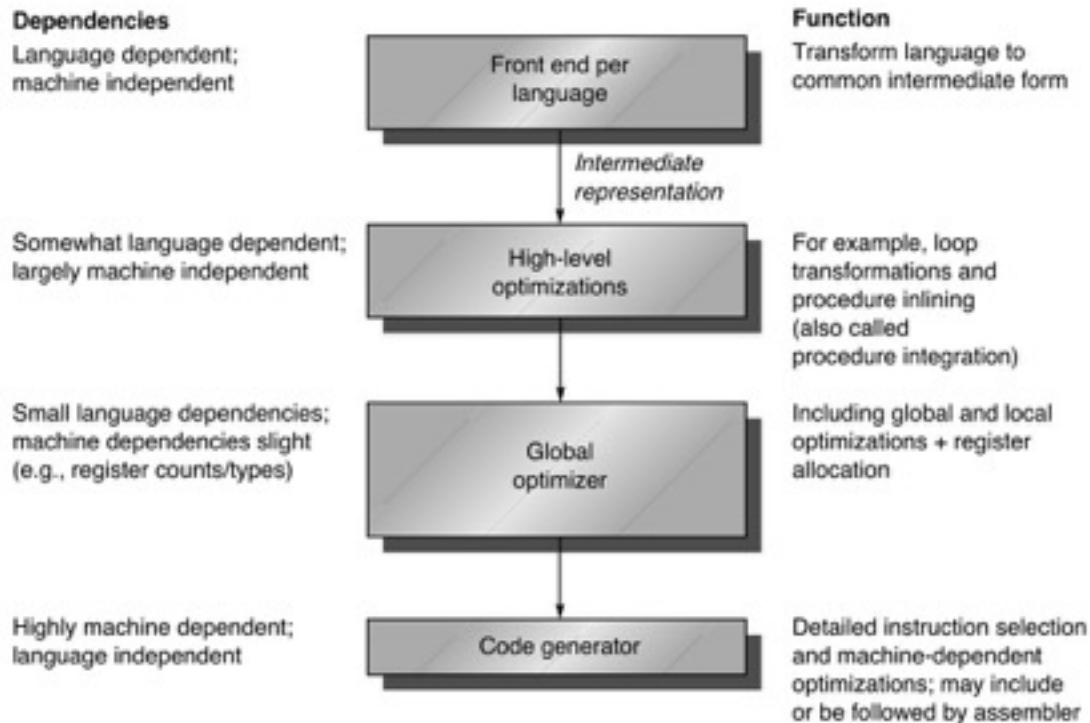
One important question is whether instructions are all encoded with the same number of bits. The options are “variable” encoding (each instruction can be a different size); “fixed” encoding (all instruction same), or a “hybrid” (where there may be a limited number of instruction sizes).

Choice influenced by competing forces:

- desire to have large # of registers + addressing modes
- code density
- ease of decoding

Optimizing Compilers (e.g., “gcc -O3”)

An optimizing compiler transforms code to make your software run faster. One important transformation is to allocate variables to registers. The number of registers impacts the effectiveness of optimizing compilers.



- Register allocation works best with ≥ 16 registers.
- Orthogonal ISA features => compiler easier to write.

Optimizing Comp

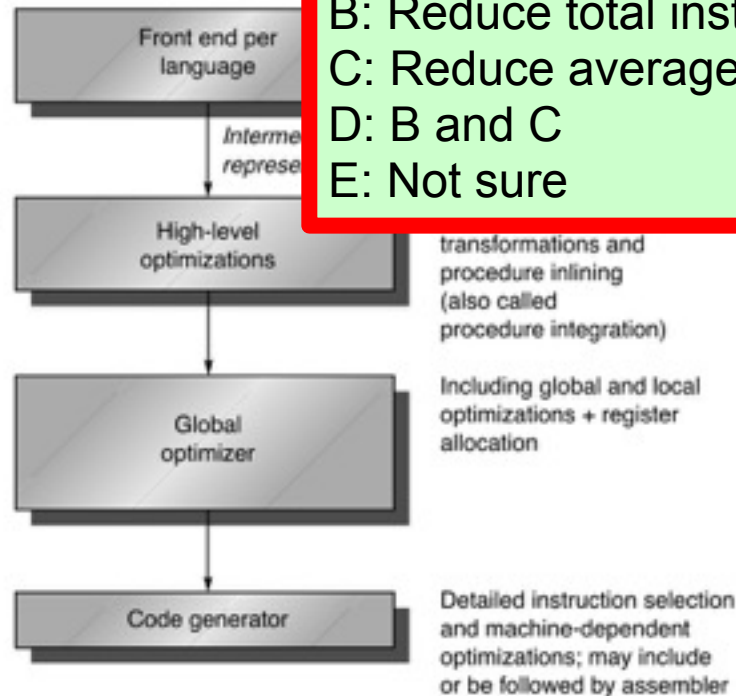
An optimizing compiler transforms code. One important transformation is to allocate a number of registers impacts the effective

Dependencies
Language dependent;
machine independent

Somewhat language dependent;
largely machine independent

Small language dependencies;
machine dependencies slight
(e.g., register counts/types)

Highly machine dependent;
language independent



Recall the processor performance equation:

$$\text{Execution Time} = \text{IC} \times \text{CPI} \times \text{cycle_time}$$

How might a compiler improve performance?

- A: Increase clock frequency
- B: Reduce total instructions executed
- C: Reduce average CPI
- D: B and C
- E: Not sure

- Register allocation works best with ≥ 16 registers.
- Orthogonal ISA features => compiler easier to write.

Optimizing Comp

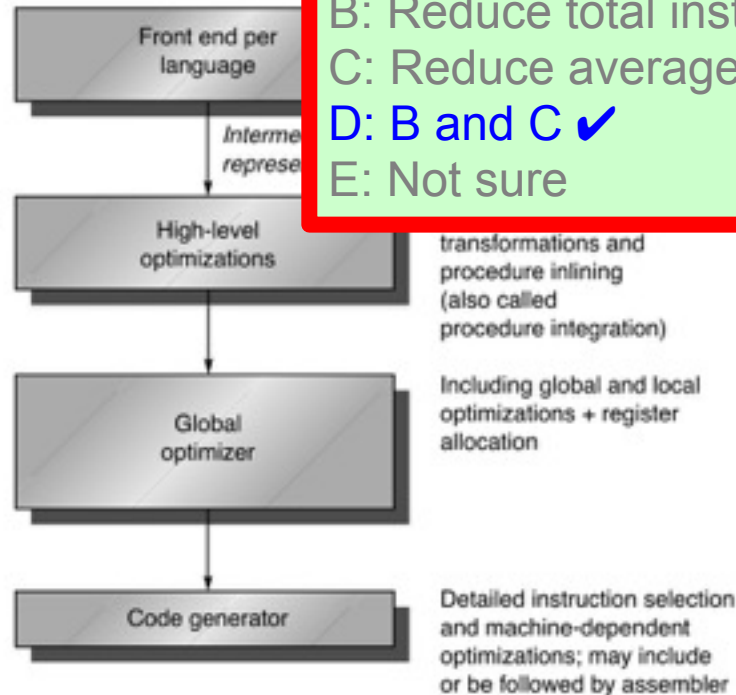
An optimizing compiler transforms code. One important transformation is to allocate a number of registers impacts the effective

Dependencies
Language dependent;
machine independent

Somewhat language dependent;
largely machine independent

Small language dependencies;
machine dependencies slight
(e.g., register counts/types)

Highly machine dependent;
language independent



Recall the processor performance equation:

$$\text{Execution Time} = \text{IC} \times \text{CPI} \times \text{cycle_time}$$

How might a compiler improve performance?

A: Increase clock frequency

B: Reduce total instructions executed

C: Reduce average CPI

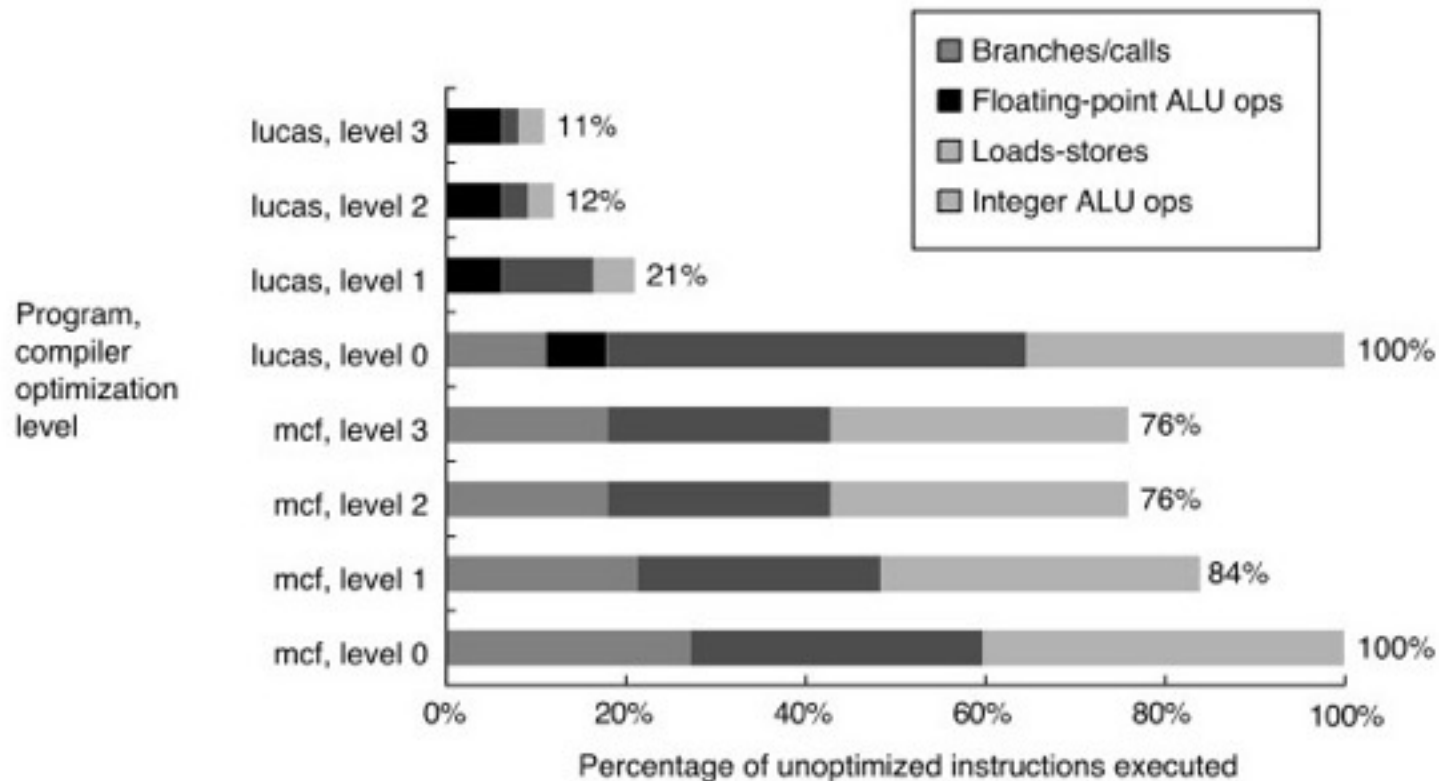
D: B and C ✓

E: Not sure

- Register allocation works best with ≥ 16 registers.
- Orthogonal ISA features => compiler easier to write.

Impact of Compiler Optimization

An important pitfall to avoid is optimizing a hardware design using software benchmarks that have not been optimized by an optimizing compiler. The data below shows how some characteristics of a program likely to be of importance to an ISA designer can change dramatically before/after using an optimizing compiler.



Putting it all together: Desirable ISA Properties...

- Local Storage
 - general purpose registers (load-store architecture)
 - at least 16 registers
- Addressing Modes
 - Displacement (12-16 bits)
 - Immediate (8-16 bits)
 - Register Indirect
- Aligned memory accesses
- Type and Size of Operands
 - 8-, 16-, 32-, and 64-bit integers, 64-bit FP
- Minimalist instruction set: Simple operations used most frequently in practice (load, store, add, sub, move, shift, ...)

Desirable ISA Properties, Cont'd

- PC-relative branches with compare equal, not-equal, less-than
- Support function calls (PC-relative, indirect), returns
- Fixed instruction encoding for performance, variable encoding for code density.
- Provide at least 16 general-purpose registers.
- Addressing modes apply to all data transfer instructions.

Summary of Slide Set #3

In this slide set we studied the components that make up all instruction set architectures. We saw that measurements of real programs were useful in informing design decisions when designing an instruction set.

We outlined some properties that are desirable and alluded to some of the reasons.

As we learn more about how an instruction set is implemented it will become more clear why a RISC instruction set is a better match to hardware (and why modern x86 uses a non-user visible RISC instruction set “internally”—known as micro-ops).

In the next slide set we start looking at not just how to build a computer that implements a given instruction set architecture, but how to make it faster.