Why registers, cont.

[H&P §2.2] Less memory traffic if values are in registers
– Program runs faster if variables are inside registers
  (compiler does “register allocation”)
– Bus can be used for other things

How many registers are needed?
Depends on:

• Compiler ability
• Program characteristics

Lots-o-registers enable two important optimizations:

• Register allocation (more variables can be in registers)
• Limiting reuse of registers improves parallelism

Reuse example:

LOAD R2, A; LOAD R3, B; LOAD R4, C; LOAD R5, D
ADD R1, R2, R3
ADD R2, R5, R4

What’s wrong with this?

Compare with …

ADD R1, R2, R3
ADD R6, R4, R5  (no reuse: had R6)

Without reuse, ADDs are “parallelizable” if there are two adders.

Instruction level parallelism (ILP) …

ILP ~ Average (CPI)$^{-1}$ ~ Number of registers
Operand access limitations

Consider again our four architecture types:

<table>
<thead>
<tr>
<th># memory operands</th>
<th># total operands</th>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>“load/store”</td>
<td>Most RISCs</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>“register/memory”</td>
<td>x86, 68000</td>
</tr>
<tr>
<td>2, 3</td>
<td>2, 3</td>
<td>“memory/memory”</td>
<td>VAX</td>
</tr>
</tbody>
</table>

**Exercise:** Here are a number of advantages (+) and disadvantages (−) of the various formats. Your task is to classify them as to whether they apply to—

Load/store (0, 3)
Register/memory (1, 2), or
Memory/memory (2, 2) or (3, 3)
(–) Destroys source operand (e.g., ADD R1,R2)
(–) Fixed-length instructions are wasteful. More instructions and lower instruction density leads to larger programs.
(–) High memory traffic (memory bottleneck)
(–) Higher instruction count (IC).
(–) CPI varies by operand location.
(–) Encoding register & memory location in 1 instruction may restrict # of registers.
(+ ) Fixed-length instructions possible: easy fetch/decode.
(+ ) Most compact (code density)
(+ ) No need for extra loads
(+ ) Simpler hardware: efficient pipeline & potentially lower CT
(+ ) Instructions take similar #s of “clocks” to execute.
(+ ) Simpler code-generation model.
(+ ) Yields good code density

Here is our classification of these reasons.

Load/store (0, 3):

Register/memory (1, 2):
Memory/memory (2, 2) or (3, 3):

**Alignment**

[H&P §2.3] Byte alignment

- Any access is accommodated

Word alignment

An access to an object of size $s$ bytes at address $A$ is aligned iff

$$A \mod s = 0$$

- Only accesses that are aligned at natural word boundaries are accommodated due to DRAM/SRAM organization
- Reduces number of reads/writes to memory
- Eliminates hardware for alignment (typically expensive)
- Often handle misalignment via software:
  - Compiler detects & generates appropriate instructions
  - …or O/S detects and runs “fixit” routine

Asking for words beginning at 0 or 4 is OK
Asking for other words requires two reads (e.g., ask for word starting at 2)
Byte ordering

Suppose that our processor loads a 32-bit word. It picks up 4 bytes from memory, say from addresses \( i, i+1, i+2, \) and \( i+3. \)

What order do these bytes appear in the register?

- Is byte \( i \) loaded into the most-significant end of the register?
- Is byte \( i \) loaded into the least-significant end of the register?

Another way of putting it: Is the low-order byte of the number stored at the lowest address, or the highest address?

“Little Endian” means that the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. (The little end comes first.) For example, a 4 byte LongInt

<table>
<thead>
<tr>
<th>Byte 3</th>
<th>Byte 2</th>
<th>Byte 1</th>
<th>Byte 0</th>
</tr>
</thead>
</table>

will be arranged in memory as follows:

<table>
<thead>
<tr>
<th>Base address + 0</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base address + 1</td>
<td>Byte 1</td>
</tr>
<tr>
<td>Base address + 2</td>
<td>Byte 2</td>
</tr>
<tr>
<td>Base address + 3</td>
<td>Byte 3</td>
</tr>
</tbody>
</table>
Intel processors (those used in PC’s) use Little Endian byte ordering.

“Big Endian” means that the high-order byte of the number is stored at the lowest address, and the low-order byte at the highest address (the big end comes first). The same LongInt would be stored as:

<table>
<thead>
<tr>
<th>Base address + 0</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base address + 1</td>
<td>Byte 2</td>
</tr>
<tr>
<td>Base address + 2</td>
<td>Byte 1</td>
</tr>
<tr>
<td>Base address + 3</td>
<td>Byte 0</td>
</tr>
</tbody>
</table>

Sun and Motorola processors (those used in Macintoshes) use Big Endian byte ordering. The Power PC is “bi-endian” because it can understand both orderings. (The operating system determines which ordering is in use; see http://www.byte.com/art/9509/sec12/art1.htm).

When does byte ordering matter? Consider the following possibilities.

- When writing programs that use any numbers (integers, floating-point, etc.).
- When writing assembly-language programs that use any numbers (integers, floating-point, etc.).
- When writing data on one computer and reading it back on another.
  - When writing integer data on one computer and reading it back on another
  - When writing floating-point data on one computer and reading it back on another
  - When writing text data on one computer and reading it back on another.
- When writing programs that write data as bytes and read it back as words.
• When writing programs that write misaligned integer data and read it back as aligned integer data (e.g., write a `longInt` to address \(i\) and read a `longInt` from address \(i+1\)).

Software tools exist for “byte-swapping,” or switching between the two byte orderings. For example, the Unix system has a built-in command called `dd`, which has an option for swapping data bytes.

*Which ordering is better?*

Advantages of Little Endian:

• Machine instructions for picking up a 1-, 2-, 4-byte, or longer number proceed in exactly the same way for all formats: first pick up the lowest order byte at offset 0.

• Multiple-precision math routines are easier to write, due to the 1:1 relationship between address offset and byte number (offset 0 is byte 0).

Advantages of Big Endian:

• Since the high-order byte comes first, you can always test whether the number is positive or negative by looking at the byte at offset zero. You don't have to know how long the number is, nor do you have to skip over any bytes to find the byte containing the sign information.

• Strings appear forwards in registers, making it easier to compare them. With Little Endian ordering, strings appear “sdrawkcasb” (backwards) in registers.

• Numbers are stored in the order in which they are printed out, so binary-to-decimal routines are particularly efficient.

For example, in Little Endian, the logical numeric value \(1234\_{16}\) (decimal 4660) is stored as \(3412\_{16}\).

A 4-byte integer is even more interesting: \(12345678\_{16}\) (logical order) becomes \(56781234\_{16}\) by swapping the high
and low 2-byte words. But this is just an intermediate step; we need to swap the bytes within the two 2-byte words, giving 78563412_{16}.

The opcode

Some computers have a fixed-length opcode. *Examples:* IBM 370, VAX, MIPS, Alpha.

Others have a variable-length opcode. *Examples:* Motorola processors (2 to 16 bits), Pentium (all of the first byte and possibly some bits from the second byte).

What’s an advantage of fixed-length opcodes?

What’s an advantage of variable-length opcodes?

*Example:* A machine with

- 16-bit instructions
- 4-bit address fields (e.g. selecting from 16 registers)

One possibility: 16 three-address instructions.

Another possibility: 15 three-address instructions.
14 two-address instructions.
31 one-address instructions.
16 zero-address instructions.

For fifteen instructions, the first four bits are the opcode.

For the other instructions, the first 4 bits are only part of the opcode.
4-bit opcode

0000  xxxx  yyyy  zzzz
0001  xxxx  yyyy  zzzz
0010  xxxx  yyyy  zzzz
0011  xxxx  yyyy  zzzz

15 three-address instructions

8-bit opcode

1111  0000  yyyy  zzzz
1111  0001  yyyy  zzzz
1111  0010  yyyy  zzzz

14 two-address instructions

12-bit opcode

1111  1110  0000  zzzz
1111  1110  0001  zzzz

31 one-address instructions

16-bit opcode

1111  1111  1111  0000
1111  1111  1111  0001
1111  1111  1111  0010

16 zero-address instructions