$AD = x_3 \cdot 3 \cdot 12$

$S + \text{reset} \cdot x_3 \rightarrow 12$

$\text{dist} = 5 - c - r - s$

$\text{cur} \leftarrow \text{dist} \cdot (c < c) \cdot (c = c) \cdot \text{reset}$

$\text{cur} \leftarrow \text{dest} \cdot (c < c) \cdot (c = c) \cdot \text{reset}$

In assembly, set $X0 \rightarrow X1$

Review Problem 2
Example: Take bits 6-4 of X0 and make them into 2-0 of X1, zeros otherwise:

```
41 = X0 // 41 = X0
AND X0, X1, #7
AND X0, X1, X2
ADD X0, X1, X2
ADD X0, X1, #100
ADD X0, X1, X2
```

Shift: Left & right logical (LSL, LSR)

Immediate

Logical: AND, ORR, EOR

Immediate (one input a constant)

Mathematic: ADD, SUB, MUL, SDIV

(Note: Just subset of all instructions)

---

Basic Operations

---

\#35

---
"Byte addressing" means that the index points to a byte of memory. A memory address is an index into the array viewed as a large, single-dimension array, with an address.
Double-words and words are aligned
264 bytes with byte addresses from 0 to 264-1
261 double-words with byte addresses 0, 8, 16, ... 264-8
260 words with byte addresses 0, 8, 16, 24

Registers hold 64 bits of data

Memory Organization (cont.)
ARM: can do either: this class assumes Little-Endian.

Intel x86, DEC VAX, Altera Nios II, Z80
Little Endian: Address of least significant byte = doubleword address
Motorola 68k, MIPS, IBM 360/370, Xilinx Microblaze, Sparc
Big Endian: Address of most significant byte = doubleword address

Little Endian

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Big Endian

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Doubleword: 263..256 255..248 247..240 239..232 231..224 223..216 215..28 0

Addressing Objects: Endian and Alignment
new ed structures (the "heap" from end. We ignore that here for simplicity)

(Note: real compilers place local variables (the "stack") from beginning of memory)

| 1047 | 1048 | 1029 | 1028 | 1027 | 1026 | 1025 | 1024 | 1023 | 1022 | 1021 | 1020 | 1019 | 1018 | 1017 | 1016 | 1015 | 1014 | 1013 | 1012 | 1011 | 1010 | 1009 | 1008 | 1007 | 1006 | 1005 | 1004 | 1003 | 1002 | 1001 | 1000 |
|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |
| 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |
| 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |
| 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |
| 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |

1032 = 4 * 256 + 8
1001 = 3 * 256 + 233

```java
x = new int[10];
y = new char[4];
for (char c = 'a'; c < 'g'; c++) {
    // c = ASCII 71
    // Array: Sequence of locations
    // Integers: 64 bits (word)
    // Characters: 8 bits (byte)
    // Pointer: Address (64 bits)
    // Array: Sequence of locations
    // Integers: 64 bits (word)
    // Characters: 8 bits (byte)
    // Pointer: Address (64 bits)
}
```

Data Storage
Note: LDRB & STRB Load & Store bytes

LDRB: 144: 66  24: 723

STUB: X3: 14 X2: 723 X1: 130 X0: 66

Registers

Memory

\[ 14 \times 3 = 24 \]
\[ 130 + 14 = 144 \]
\[ \text{memory} = [x_{3+20}] \]

Load & Stores

Move data between memory and registers

All operations on registers, but too small to hold all data
What would the results of this C++ code be in memory? Assume we start using memory at 1000.

Review Problem 4
Load the following characters

The address of the start of a character array is stored in X0. Write assembly to

Addressing Example
Array Example

\[
V_{ck+1} = \text{new} [x0 + x1 + 8] \\
V_{ck} = \text{new} [x0 + x1 + 8]
\]
Execution Cycle Example

Note: IR: Instruction Register
PC: Program Counter

Instructions are 32b
Word addresses

execution
Most commonly used are substractions, so we have a synonym: CMP

ADDs, ADDPs, ANDs, ANDPs, SUBs, SUBPs, some floating point.

Operations that set the flag register contents:

Carry: was the carry-out true?

Overflow: was result magnitude too big to fit into 64-bit register?

Zero: was result 0?

Negative: was result a negative number?

Flag register holds information about result of recent math operation.

Flags/Condition Codes