Loading and Storing Operands in the ARM processor

Background

The ARM processor uses what is known as a "*load-store*" architecture. This means that the only instructions that can reference memory are loads and stores (i.e., LDR, STR and their variants); a computation like $a = b + c$ must first use *load* instructions to read the operands (b and c) from memory into registers, add the contents of the two registers and leave the result in a register, and then use a *store* instruction to write the result back to memory. In other words, instructions such as add, subtract, bitwise-and, etc., can only operate on the contents of registers; they cannot operate directly on the contents of memory locations.

Logically, memory is simply a collection of 8-bit bytes, with each byte having its own unique memory address. An address is a 32-bit number that can select one of 2^{32} different bytes (4 gigabytes) in memory. Adjacent bytes are combined to form 16-bit half words, 32-bit words and 64-bit double words. The ARM processor uses the *Little Endian* numbering convention, so that the address of these larger operands is always specified as the address of their least-significant byte. For example, the 16-bit half word whose 32-bit address is 104, consists of two bytes located at addresses 104 and 105.

Physically however, memory uses a 32-bit data bus so that it can transfer as many as four bytes in a single read or write cycle[1]. Even though four bytes are retrieved, sometimes less than four bytes are actually used, either because the desired operand is a byte or a half word, or because the operand is misaligned – i.e., split across two adjacent 32-bit memory words. Misaligned or double word operands require more than one memory cycle.

Storing a result into memory

All registers in the ARM processor are 32 bits wide. When writing a byte or half word result into memory, all four bytes of the register are placed on the 32-bit memory data bus, but only one or two of those bytes are enabled to be written during the write cycle.

For example, consider what happens when writing a half word result to address 104: The two memory bytes that should be written are at addresses 104 and 105. However, the 32-bit memory data bus is actually four bytes wide, corresponding to addresses 104, 105, 106 and 107. Each byte of the memory data bus has its own write enable; during the memory write cycle, the write enable signals for addresses 106 and 107 are disabled so that only the bytes at addresses 104 and 105 are modified. The particular type of store instruction determines which bytes of the 32-bit register are written:

Instruction	*Operand*	*Memory Byte Locations Actually Written*			
STRB	8-bits				Address N
STRH	16-bits			Address N+1	Address N
STR	32-bits	Address N+2	Address N+2	Address N+1	Address N
STRD	Lower 32 bits	Address N+2	Address N+2	Address N+1	Address N
	Upper 32 bits	Address N+7	Address N+6	Address N+5	Address N+4

Loading an operand from memory to a register

When loading an 8-bit byte or 16-bit half word from memory into a 32-bit register, we must choose a load instruction according to the size of the operand *and* how the processor should fill the other bits of

[1] Physically, the memory is organized as 2^{30} 32-bit words. The most-significant 30 bits of the address select the word, while the remaining bits are used to select the appropriate byte(s) within that word.

the destination register. The requirement is that the numerical value represented by the operand must be preserved. Thus the operand itself is always right-aligned within the register, and the most-significant bits of the register filled according to whether the operand is signed or unsigned.

Unsigned operands must fill the extra bit positions of the register with zeroes so that the result represents the same value as the smaller memory operand. That's called "zero-filling", and is what the LDRB and LDRH instructions do:

Instruction	*Operand*	*Contents of 32-bit Destination Register*		
LDRB	8-bits	**24 bits of zeroes**		Address N
LDRH	16-bits	**16 bits of zeroes**	Address N+1	Address N

For signed operands, the extra bit positions of the register must be filled with copies of the sign bit of the smaller memory operand[2]. That's called "sign-extending", and is what the LDRSB and LDRSH instructions do:

Instruction	*Operand*	*Contents of Destination Register*		
LDRSB	8-bits	**24 copies of the sign bit**		Address N
LDRSH	16-bits	**16 copies of the sign bit**	Address N+1	Address N

Of course zero-filling and sign-extension are irrelevant when loading a 32-bit word or 64-bit double word, so there is only one instruction (LDR) for loading a 32-bit word operand and only one (LDRD) for loading a 64-bit double word operand.

Pointers

When translating C into ARM assembly, the size of an operand is determined by its data type. Variables declared as type char are stored as 8-bit bytes, short ints as 16-bit half words, long ints as 32-bit words, and long long ints as 64-bit double words. Since pointers always contain an address, they must always be 32-bits wide – regardless of the size of the objects they point to. Even if the pointer is declared to point to (say) a char, the pointer itself is 32-bits wide. Therefore the pointer is loaded with an LDR instruction and stored using an STR. However, when the pointer is de-referenced, the corresponding load or store must use an instruction that matches the data pointed to by the pointer. For example:

```
char c ;           // an 8-bit byte
char *p ;          // a 32-bit pointer

p = &c ;           // copy the 32-bit address of c into p

      ADR   R0,c         ; get the 32-bit address of c
      STR   R0,p         ; store it into p

*p = 0 ;           // store a 0 into c (via p)

      LDR   R0,=0        ; get a zero
      LDR   R1,p         ; get the 32-bit address of destination
      STRB  R0,[R1]      ; store the 8-bits of zeroes into c
```

[2] "Sign-extension" is used because the ARM processor (like most modern processors) uses the two's complement representation of signed numbers. Other representations would require other techniques for filling the extra bit positions.