Doing C-style structs on cell addressed uCore

Klaus Schleisiek - kschleisiek@wauland.de

Last year, the technical high-school of Windisch (FHNW - Fachhochschule Nordwest-Schweiz) realized a uCore back end for LCC (Little C-Compiler). LCC was enhanced by Markus Knecht to become FCC (Forth C-Compiler) integrating advanced stack allocation techniques into the front end. This substantially reduced the number of local variables on the return stack and turns uCores dual stack architecture into a performant C engine.

Another problem with C is its fundamental byte orientation. I took this problem lightly for a long time proposing to declare a byte to be any number of bits as long as it is more than eight. Unfortunately, this way of looking at things does not help in the case of C at all: Unions may be defined to access a quadruple of bytes as one 32-bit integer.

Therefore, bytes need to be accessible within larger memory cells - of which only even multiples of eight make sense at all. So lets discuss a 32-bit word width architecture. Integer (32-bit **i@**, **i!**), word (16-bit **w@**, **w!**), and byte (8-bit **c@**, **c!**) accesses within a 32-bit cell are needed.

For fetches this is easy. **i@**, **w@** and **c@** can be realized in a single cycle, perhaps followed by the word **signed** that takes care of appropriate sign extension. Without signed, the most significant bits will be zero filled. Stores are more complicated requiring an un-interruptible dual cycle read-modify-write cycle. We fetch the appropriate 32-bit cell, modify the byte or word to be written and write the result back to the cell.

This leaves us with two more problems: 1) how to do byte/word addressing and 2) what to do when access happens to a "misaligned" address. The answer to 2) is classical: We raise an exception and execute a call to the "misaligned address trap" address. More on this later.

1) is more tricky and there have been two approaches to addressing bytes on cell based machines. In a 32-bit machine, we need two additional bits to locate a byte. We observe that this reduces the address space of the word addressed machine by a factor of four, which is not a real limitation on a 32-bit machine, and if it is, upgrade to 64-bits.

This leaves the question: Where do we put the additional address bits? The most intuitive solution is to shift the word address two times to the left and use the two new least significant bits for selecting a byte within a 32-bit cell. Byte address arithmetic is trivial - normal 2scomplement arithmetic will do. But unfortunately, under this approach a 32-bit integer address is a completely different number than a 32-bit cell address accessing the same memory cell.

Therefore, another solution turns out to be more efficient over all: The two additional bits are placed in the two most significant bits of a byte address. This way, **i@** as well as **@** operating on the same numerical value will access the same memory cell as long as the two most significant bits are zero. But how do we do byte address computation? All we need is just one operator **byte+ (caddr n -- caddr')** that adds n, a signed number of bytes, to the byte or cell address on the stack. All the pains of doing weird arithmetic on a number whose least significant bits are kept in the two most significant bit positions are encapsulated in the **byte+** operator. On uCore, this is a single cycle instruction.

Now the last problem to be solved is the behaviour of the "misaligned address trap". A call to this trap temds to be the result of a software bug. Most of the time, we could just replaced the misaligned address by the nearest properly aligned address and the software will work as expected. Therefore, a basic misaligned trap handler should correct the address and re-execute the trapped memory access instruction. On the side, it can do statistics so the programmer is able to learn about his software bugs after the program executed.

These are the new words introduced:

i@ (caddr -- 32b)
fetches a 32-bit number from memory address caddr. If the two most significant bits of caddr are non-zero, the misaligned address trap will be called.

w@ (caddr -- 16b)
fetches a 16-bit number from memory address caddr. The 16 most significant bits of 16b will be zero. The most significant bit of caddr determines, which 16-bit section of the cell located at the equivalent cell address of caddr will be selected. If the second but most significant bit of is non-zero, the misaligned address trap will be called. As a side effect, the "word" status flag will be reset to zero.

c@ (caddr -- 8b)
fetches an 8-bit number from memory address caddr. The 24 most significant bits of 8b will be zero. The two most significant bits of caddr determine, which 8-bit section of the cell located at the equivalent cell address of caddr will be selected. As a side effect, the "word" status flag will be set to one.

signed (u -- n)
Depending on the state of the "word" status flag, u will be sign extended. If the "word" status flag is set, bit 7 of u will be copied into bits 8 to 31 of n. Otherwise, bit 15 of u will be copied into bits 16 to 31 of n.

i! (n caddr --)
stores n into the memory cell at caddr. If the two most significant bits of caddr are not zero, the misaligned address trap will be called.

w! (16b caddr --)
stores 16b into the memory cell at caddr. This is an uninterruptible read-modify-write cycle, because 16b has to be merged with the 32-bit content of the memory cell at caddr. If the most significant bit of caddr is set, 16b will be stored into bits 16 to 31 of the memory cell at caddr. Bits 0 to 15 will not be affected. If the second but most significant bit of caddr bit is non-zero, the misaligned address trap will be called.

c! (8b caddr --)
stores 8b into the memory cell at caddr. This is an uninterruptible read-modify-write cycle, because 8b has to be merged with the 32-bit content of the memory cell at caddr. The two most significant bits of caddr determine, which 8 bit section of the memory cell at caddr will be modified; the remaining bits will not be affected.

MSB setting	destination for 8b
00	bits 0 to 7
01	bits 8 to 15
10	bits 16 to 23
11	bits 24 to 31.

byte+ (caddr n -- caddr')
performs byte address arithmetic on caddr. This is different from standard +, because the two least significant bits of the byte address are located in the two most significant bits of caddr.