f8

an architecture for small embedded systems

Philipp Klaus Krause

2026-01-31

8/16-Bit architectures

- In between low-end (4-bit) and high-end (32- and 64-bit microcontrollers).
- Typically programmed in C
- Devices cost about 1¢ to 1 €
- Data memory typically in the range of a few B to a few KB
- Program memory typically a few KB
- Market dominated by proprietary architectures, and ancient architectures implemented by many vendors

The Small Device C Compiler

- Free C compiler (ANSI C89, ISO C99, ISO C11, ISO C23)
- Freestanding implementation or part of a hosted implementation
- Supporting tools (assembler, linker, simulator, ...)
- Works on many host systems (GNU/Linux, Windows, macOS, Hurd, OpenBSD, FreeBSD, ...)
- Targets various 8-bit architectures (MCS-51, DS80C390, Z80, Z180, eZ80, Rabbit, SM83, TLCS-90, HC08, S08, STM8, pdk14, pdk15, pdk13, MOS 6502, WDC 65C02)
- Has some unusual optimizations that make sense for these targets (in particular in register allocation)
- Users: µC programmers, and retrocomputing/-gaming developers

Lessons learned - big picture

- An efficient stackpointer-relative addressing is essential for reentrant functions
- A unified address space is essential for efficient pointer access
- Registers help
- Hardware multithreading can replace peripheral hardware, but it needs good support for atomics, and thread-local storage
- Irregular architectures can be very efficient with tree-decomposition-based register allocation
- A good mixture of 8-bit and 16-bit operations helps
- Pointers should be 16 bits

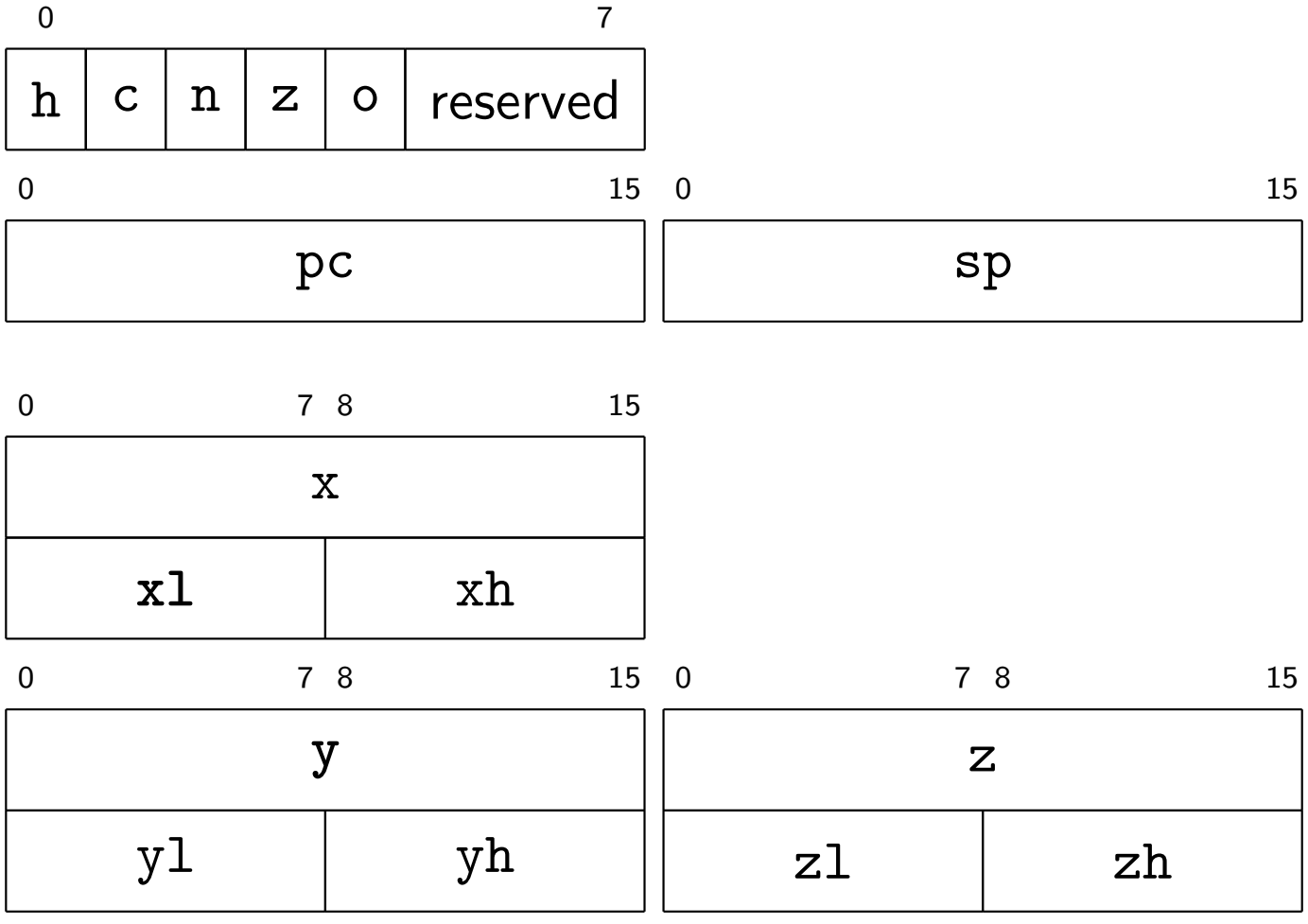
Lessons learned - details

- Zero-page, etc addressing isn't useful if we have efficient stackpointer-relative addressing
- A index-pointer-relative read instruction for both 8 and 16 bits is important
- Prefix bytes can be a good way to allow more operands (e.g. registers)
- Hardware $8 \times 8 \rightarrow 16$ multiplication helps
- Division is rare
- Multiply-and-add helps speeds up wider multiplications
- BCD support provides cheap printf without need for hardware division
- Good shift and rotate support helps

Where do we get - big picture

- 8/16 bit
- Irregular CISC
- The core becomes bigger than for RISC, but we save so much on code memory that it is worth it
- f8l instruction subset for smaller core

Register set



Example 8-bit 2-operand instruction

adc: 8-bit addition with carry

Assembler code	Operation	f8l
`adc xl, op8_2`	`xl = xl + op8_2 + c`	Yes
`adc altacc8, op8_2`	`altacc8 = altacc8 + op8_2 + c`	Yes
`adc op8_2ni, xl`	`op8_2ni = op8_2ni + xl + c`	Yes

where

`op8_2`	Any of `xh`, `yl`, `yh`, `zl`, `#i`, `mm`, `(n, sp)`, `(nn, z)`.
`op8_2ni`	Any of `xh`, `yl`, `yh`, `zl`, `mm`, `(n, sp)`, `(nn, z)`.
`altacc8`	Any of `xh`, `yl`, `yh`, `zl`, `zh`.

Rough instruction set overview

- Instruction classes: 8-bit 2-operand, 8-bit 1-operand, 16-bit 2-operand, 16-bit 1-operand, 8-bit loads, 16-bit loads, other 8-bit, other 16-bit, jumps.
- Most operate on an "accumulator" (which is both the destination and a source operand, can be changed by prefix), order of operands can be swapped by prefix.
- All instructions write at most one 16-bit register and a 16-bit memory location.

Basic safety and security features

- Watchdog
- Reads from 0x0000 trap
- Instruction 0x00 traps

Current state

- f8 port in SDCC (compiler, assembler, simulator, passes regression tests
- f8 and f8l Verilog implementations
- https://github.com/f8-arch
- https://sdcc.sourceforge.net/
- Still doing a few last optimizations on the opcode map