Hardware/Software Co-Design **MicroCore**

Ulrich Hoffmann
FH Wedel

Overview

- What is **MicroCore**?
- architecture
- systems design
- hardware/software-co-design
- future work

What is MicroCore?

MicroCore

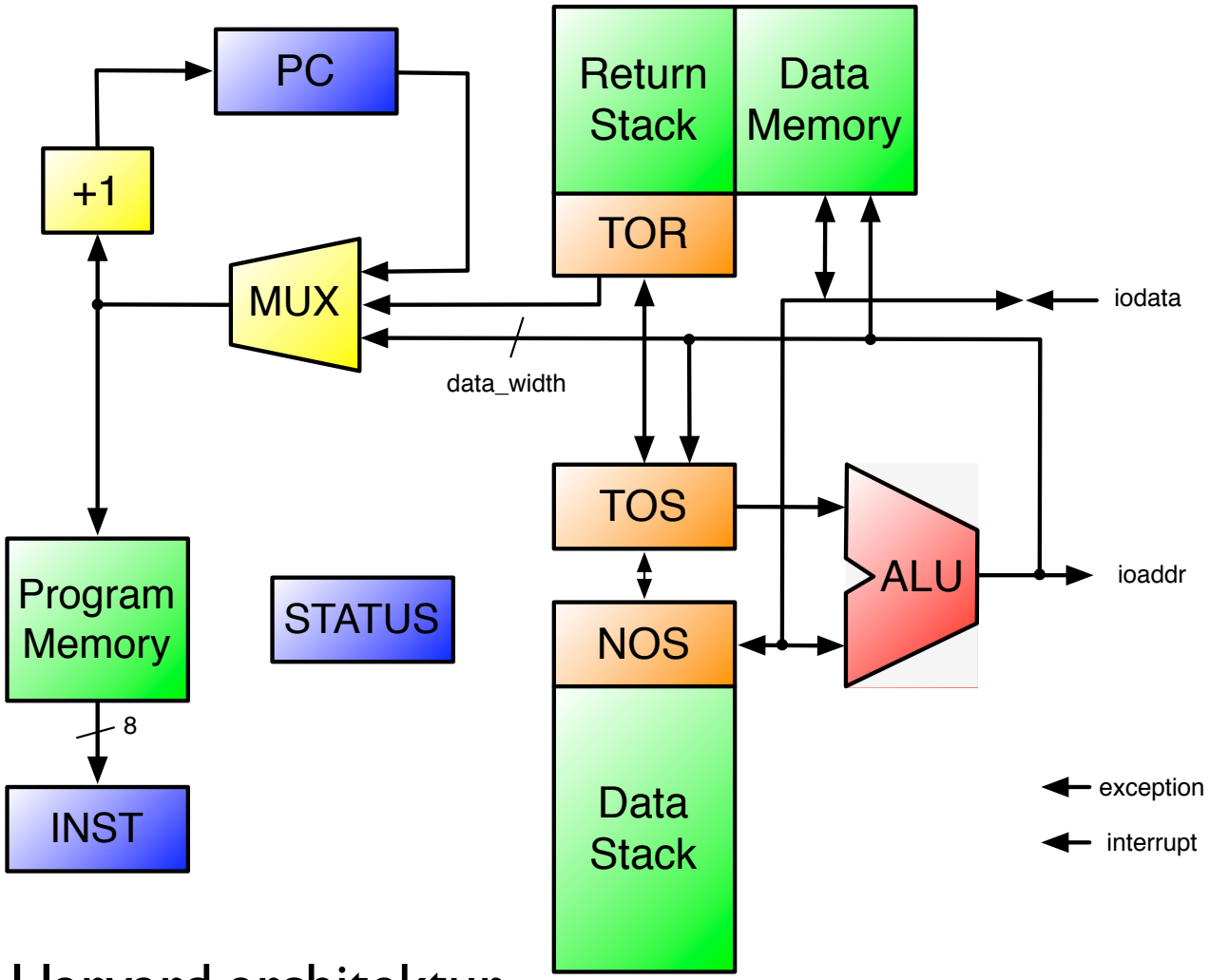
- Softcore-Mikroprozessor/Controller
- Open source in VDHL
- synthetesizable for FPGAs (Actel, Altera, Lattice, Xilinx)
- stack based Harvard architecture
- tool chain:
  - Forth cross compiler
  - realtime kernel
  - interactive debugger
  - C compiler in development

What is MicroCore used for?

- **`MicroCore`** invented by
  Klaus Schleisiek, SEND Offshore GmbH,
  Hamburg
- Used in ocean bottom seismics

- embedded systems
- device/applience programming
- made-to-measure hardware/software combination
- system understanding down to the bits
- independence from hardware vendors

MicroCore architecture



- Harvard architektur
- scalable with of data words, LITeral instruktion
- interrupts and exceptions

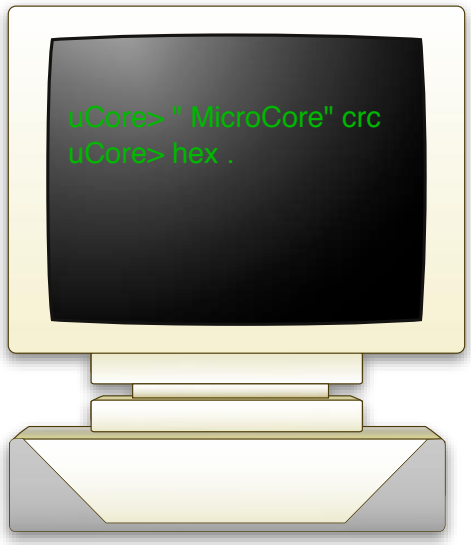
MicroCore instruction set

7	6	5	4	3	2	1	0
$80	$40	$20	$10	$8	$4	$2	$1
Lit/Op	Type		Stack		Group		

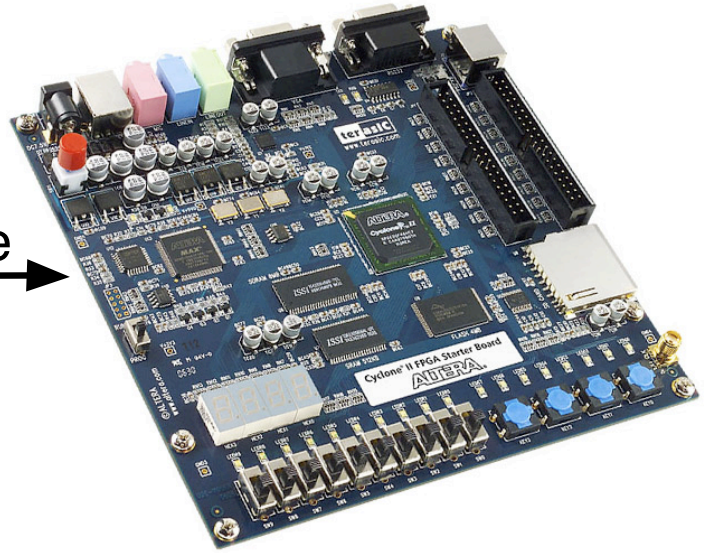
Code	Name	Action
00	BRA	Branches, Calls and Returns
01	ALU	Binary and Unary Operators
10	MEM	Data-Memory and Register access
11	USR	Not used by core, free for user extensionsUser instructions / immediate calls

Stack	act	Operation	Forth operators / phrases
none	none	Complex math instructions, see below	`SWAP`
pop		Stack -> NOS <op> TOS -> TOS	`+ - AND OR XOR NIP`
push		NOS <op> TOS -> TOS TOS -> NOS -> Stack	`2DUP_+ OVER`
both	none	TOS <uop> -> TOS Unary math instructions, see below	`0= 2* ROR ROL 2/ u2/`

interactive host/target progam development



umbilical line
RS232



interaktive debugger
w/ symbol table

monitor program

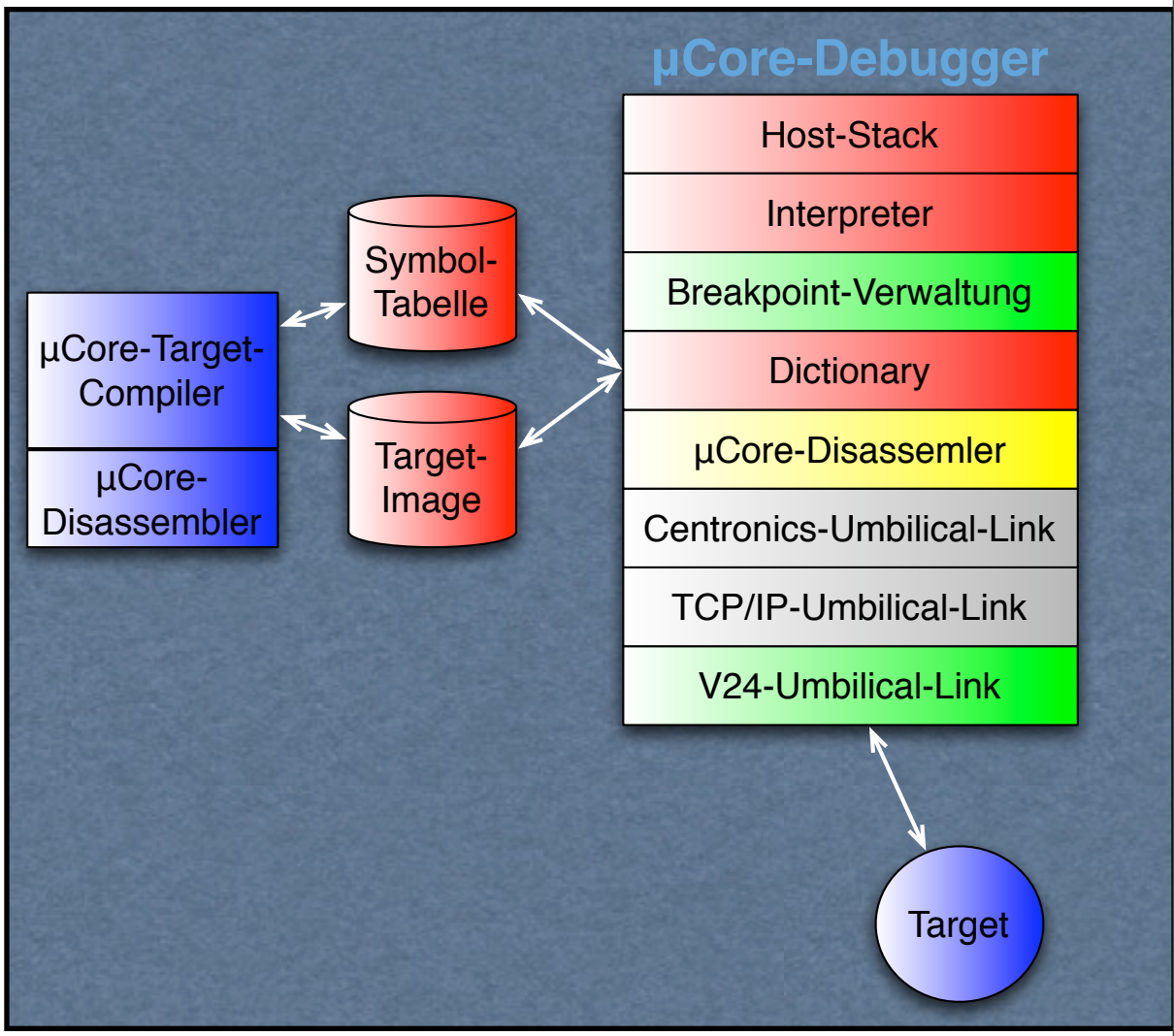
program download → load

subroutine start addresses → execute

memory content ↔

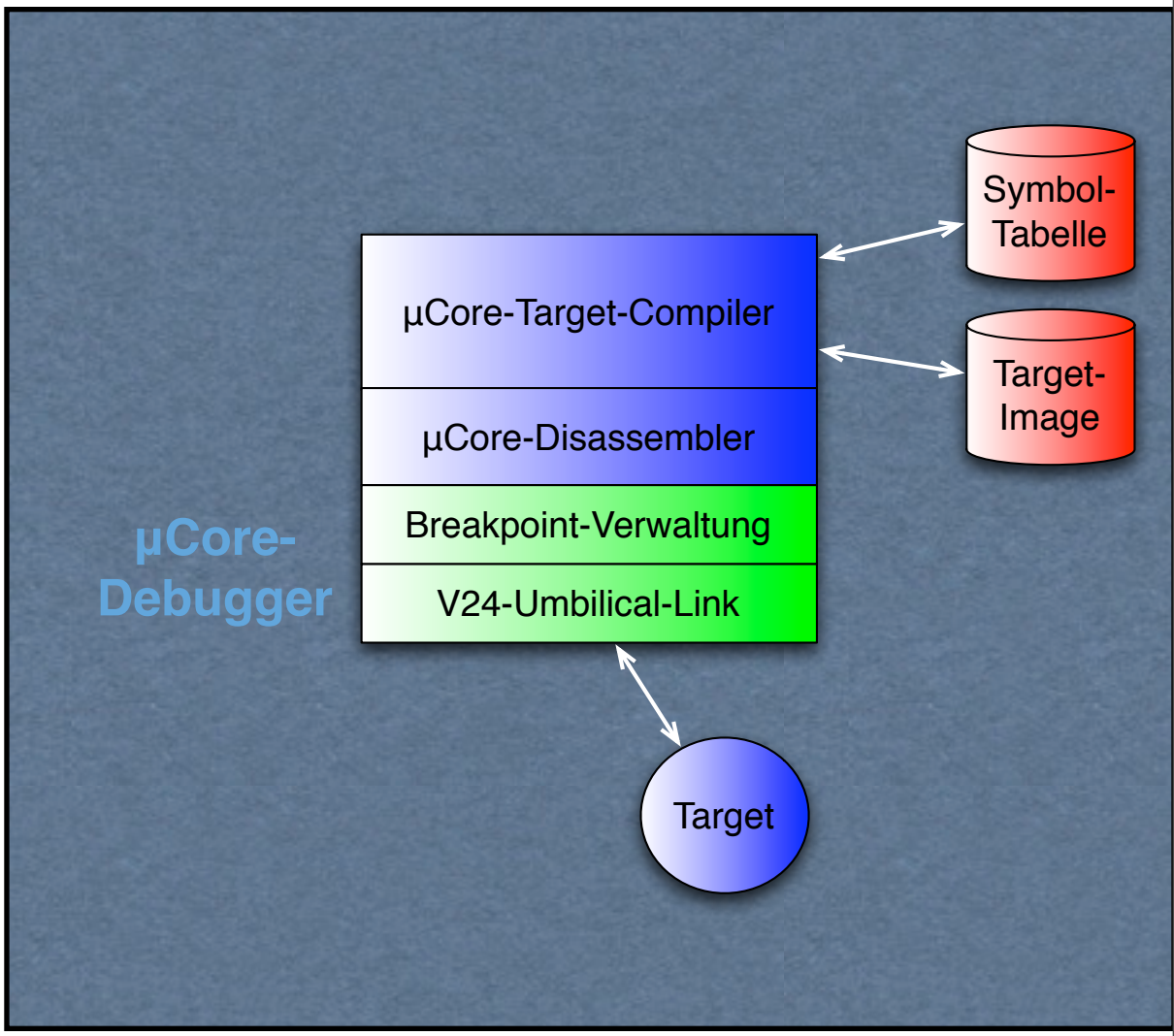
MicroCore Debugger

- old Debugger in C
  - Linux dependet (termios)
  - C99 w/ local Functions - not really portable
  - much of what Forth already has
  - much of what the target compiler already has
  - no longer required functionality

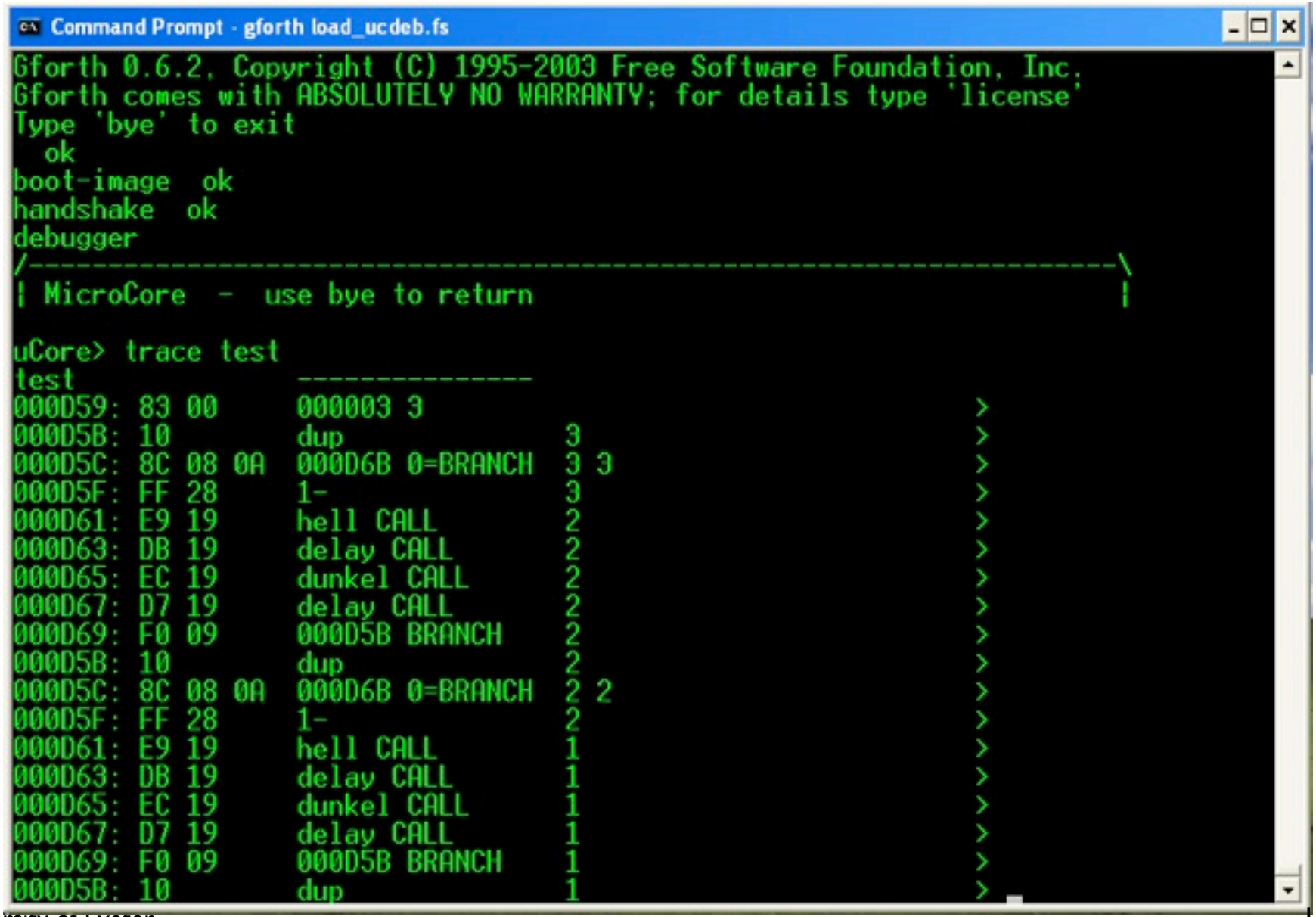


MicroCore Debugger

- ✓ **new** Debugger in Forth
  - ✓ Gforth - portable
  - ✓ currently V24 für Windows, but Code für Linux and Mac in Gforth exists
  - ✓ integrated environment
  - ✓ incrementel compilation



How does it look like?

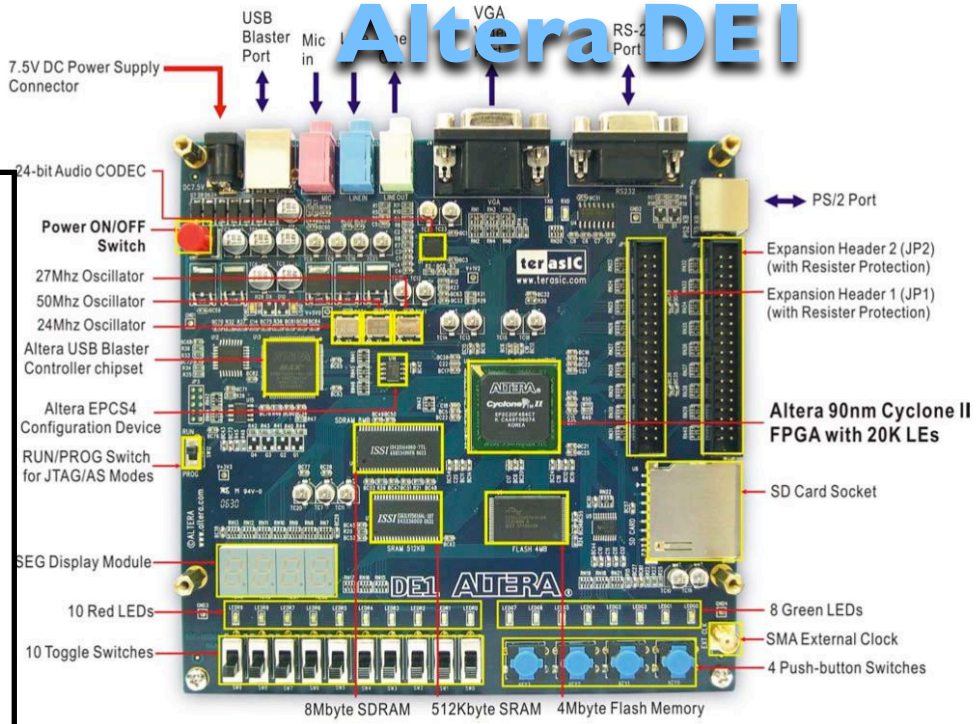


FPGA-Projekt

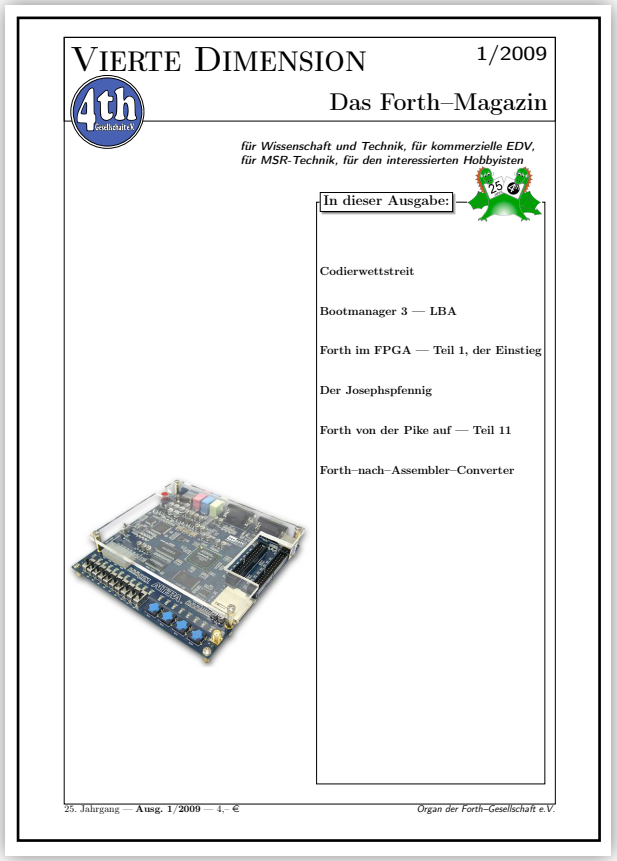
currently two target platforms:

Lattice XP2-8E

Altera DE1



FPGA article series



VIERTE DIMENSION 1/2009

Das Forth–Magazin

für Wissenschaft und Technik, für kommerzielle EDV, für MSR-Technik, für den interessierten Hobbyisten

In dieser Ausgabe:

Codierwettstreit

Bootmanager 3 — LBA

Forth im FPGA — Teil 1, der Einstieg

Der Josephspfennig

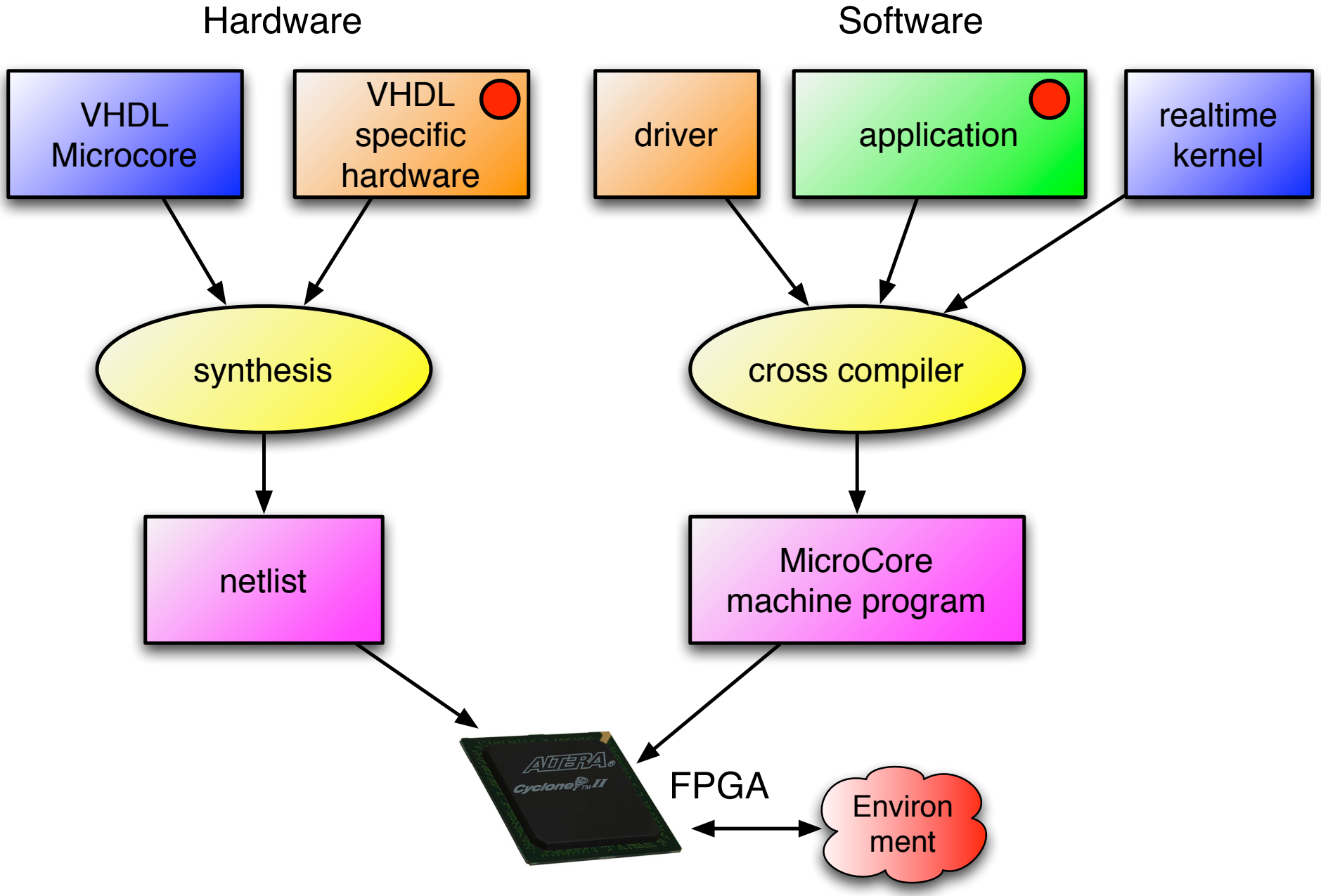
Forth von der Pike auf — Teil 11

Forth–nach–Assembler–Converter

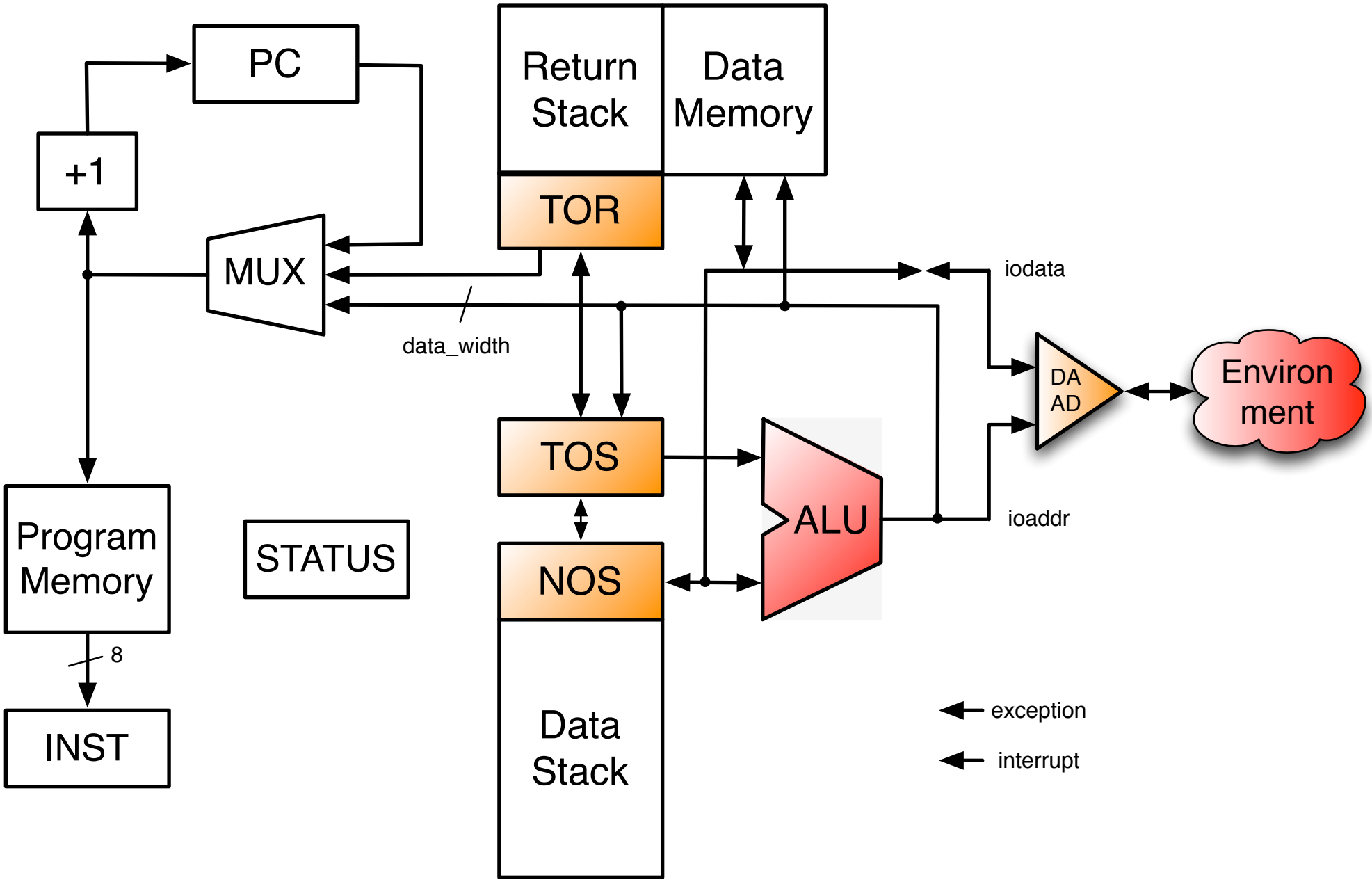
25. Jahrgang — **Ausg. 1/2009** — 4,– €

Organ der Forth–Gesellschaft e.V.

System Design

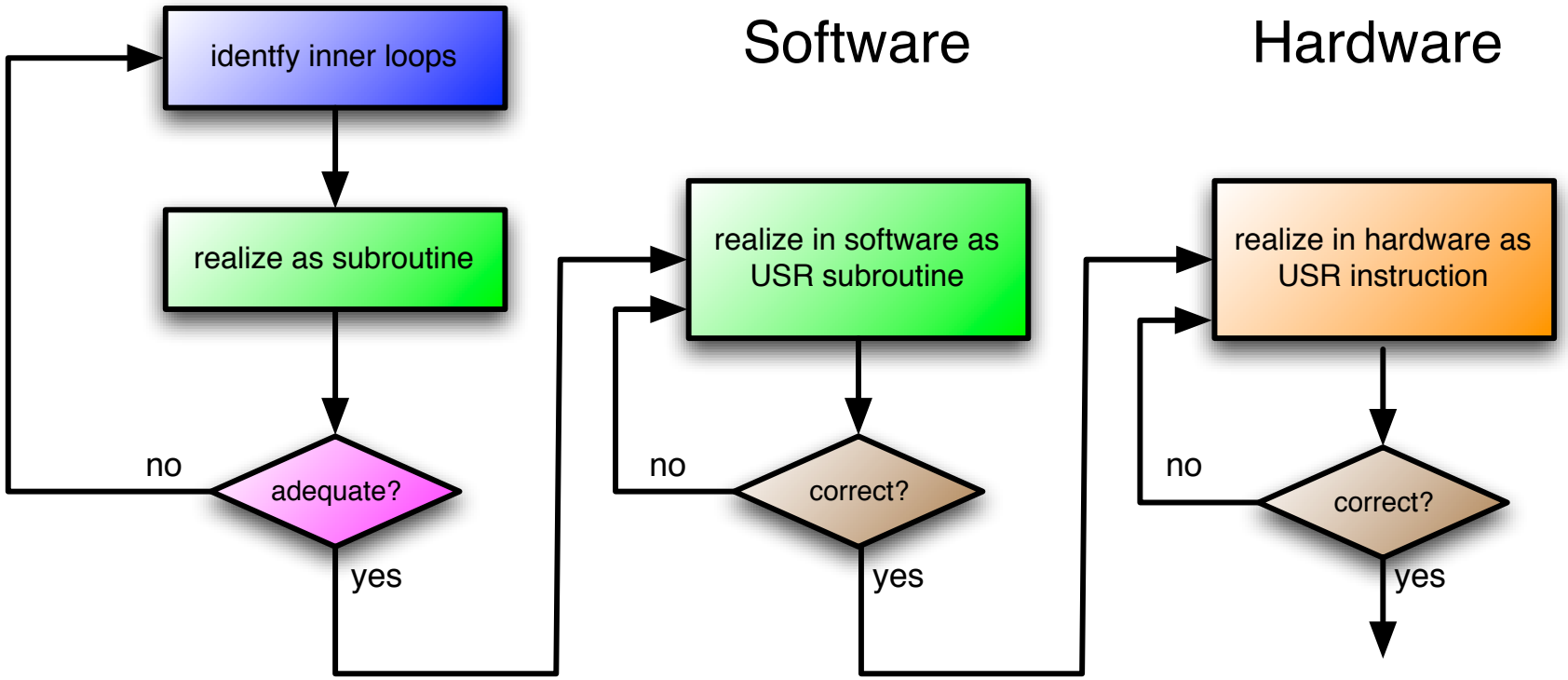


Connecting Hardware



Connecting Hardware

- alternative: user instructions
  - 32 free op codes `USR0` – `USR31`
  - standard behavior:
    - call to fixed addresses



Example: Cyclic Redundany Check

- error detection in data trans mission
- polynomial division
  - of a bit stream
  - by a generator polynomial
  - remainder is CRC value

```
11010110110000
10011
-----
 10011
 10011
 -----
  000010110
      10011
      -----
       010100
        10011
        -----
          1110 (Rest)
```

$1x^4 + 0x^3 + 0x^2 + 1x^1 + 1x^0$

- now: calculate CRC-16 according to CCITT
- polynomial: $x^{16} + x^{12} + x^5 + 1$

Quelle: wikipedia

Example: Cyclic Redundany Check

- Software realization
  - process data byte-wise:

C

```
unsigned int  crc_ccitt_step(unsigned char b
                             unsigned int crc) {

  crc = (unsigned char)(crc >> 8) | (crc << 8);
  crc ^= b;
  crc ^= (unsigned char)(crc & 0xff) >> 4;
  crc ^= (crc << 12);
  crc ^= ((crc & 0xff) << 4) << 1;
  return crc }

unsigned int crc_ccitt(unsigned char* data
                       int len) {
  unsigned int crc = 0xFFFF;
  for (int i=0; i<len; i++) {
     crc = crc_ccitt_step(data[i], crc);
  }
  return crc;
}
```

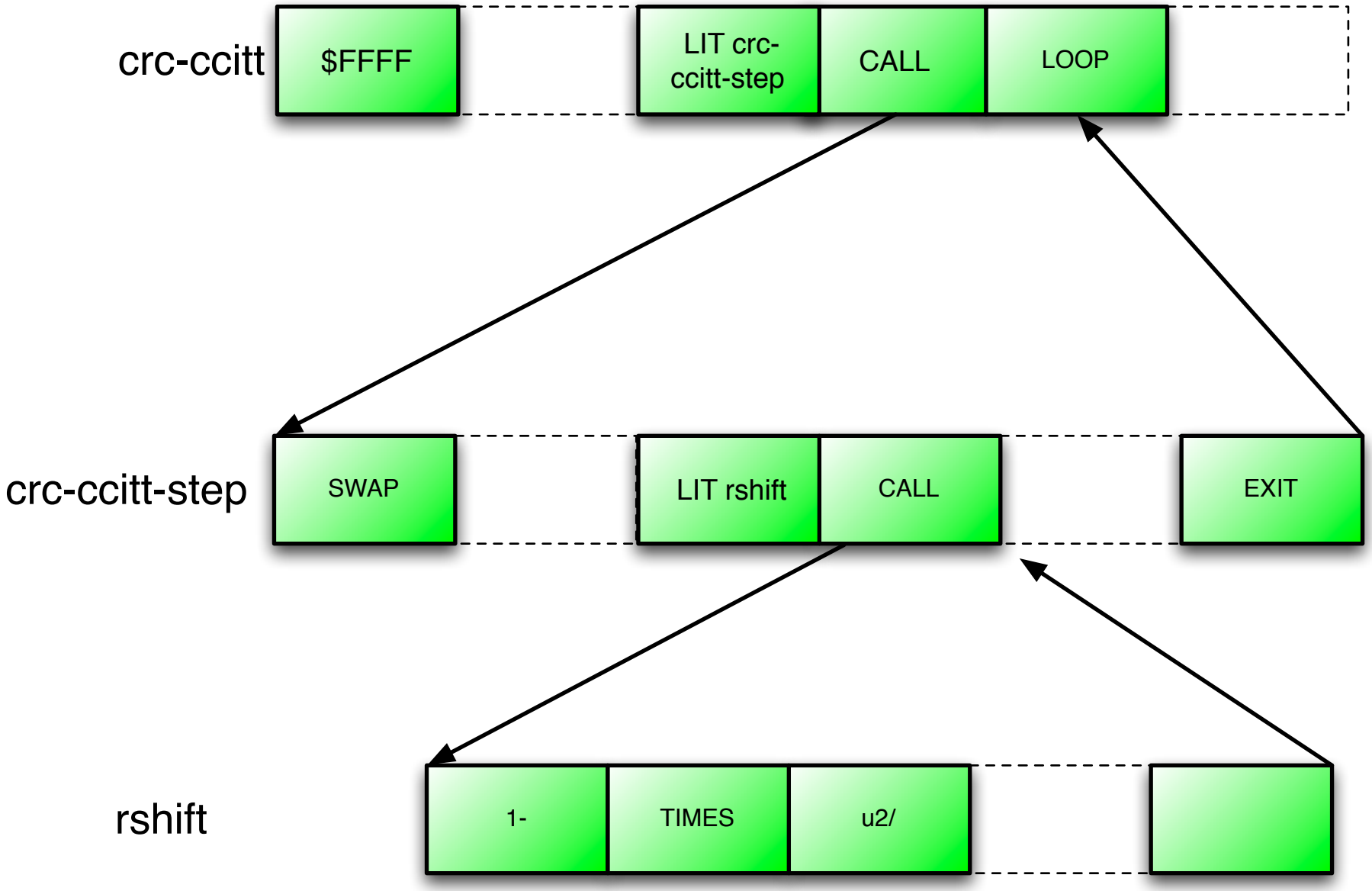
Forth

```
: lshift ( x n -- x' )  1- times 2* ;
: rshift ( x n -- x' )  1- times u2/ ;

: crc-ccitt-step ( crc b -- crc' )
  swap dup 8 lshift
  swap 8 rshift or xor
  dup $FF and 4 rshift xor
  dup 12 lshift xor
  dup $FF and 5 lshift xor
  $FFFF and
;

: crc-ccitt ( addr len -- crc )
  $FFFF rot rot
  bounds ?DO  I @ crc-ccitt-step
              LOOP
;
```

Call Structure

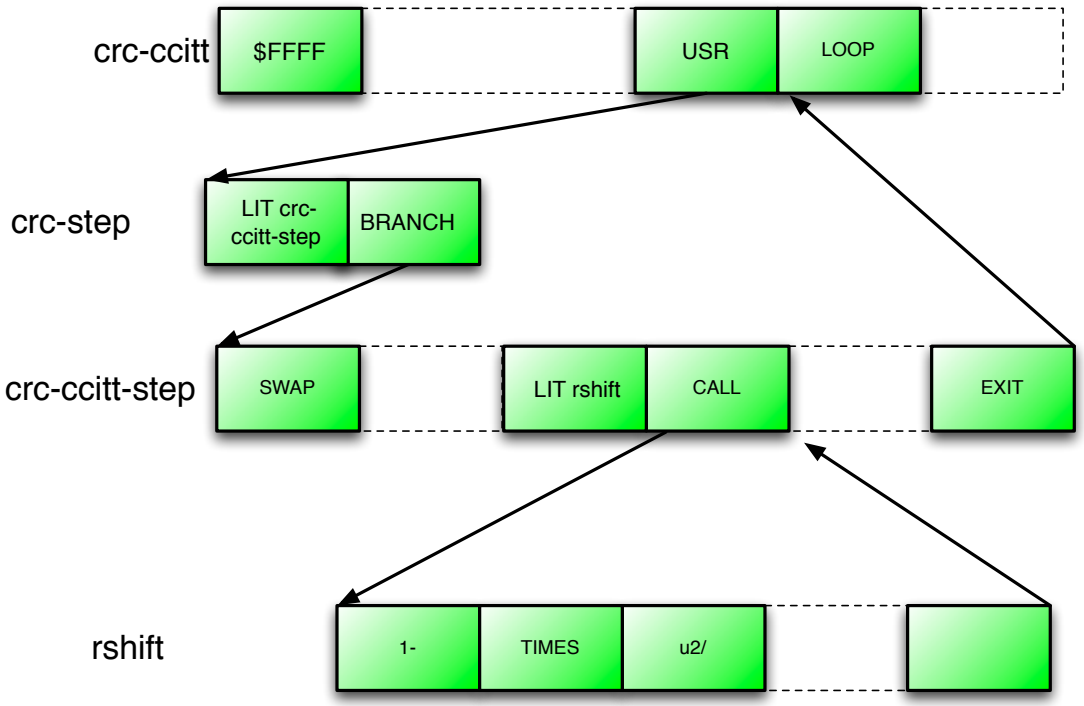


Software Realization by a USR-Instruction

```
5 USR: crc-step  ['] crc-ccitt-step nop branch ;USR

: crc-ccitt ( addr len -- crc )
  $FFFF rot rot
  bounds ?DO  I @ crc-step
              LOOP
;
```

Forth



Hardware Realization as a USR-Instruction

VHDL

```
function crc_ccitt_step
  (d: std_logic_vector(7 downto 0);
   crc:  std_logic_vector(15 downto 0))
  return std_logic_vector is

  variable result: std_logic_vector(15 downto 0);

begin
  result(0) := d(4) xor d(0) xor crc(8) xor crc(12);
  result(1) := d(5) xor d(1) xor crc(9) xor crc(13);
  result(2) := d(6) xor d(2) xor crc(10) xor crc(14);
  result(3) := d(7) xor d(3) xor crc(11) xor crc(15);
  result(4) := d(4) xor crc(12);
  result(5) := d(5) xor d(4) xor d(0) xor crc(8) xor crc(12) xor crc(13);
  result(6) := d(6) xor d(5) xor d(1) xor crc(9) xor crc(13) xor crc(14);
  result(7) := d(7) xor d(6) xor d(2) xor crc(10) xor crc(14) xor crc(15);
  result(8) := d(7) xor d(3) xor crc(0) xor crc(11) xor crc(15);
  result(9) := d(4) xor crc(1) xor crc(12);
  result(10) := d(5) xor crc(2) xor crc(13);
  result(11) := d(6) xor crc(3) xor crc(14);
  result(12) := d(7) xor d(4) xor d(0) xor crc(4) xor crc(8) xor crc(12) xor crc(15);
  result(13) := d(5) xor d(1) xor crc(5) xor crc(9) xor crc(13);
  result(14) := d(6) xor d(2) xor crc(6) xor crc(10) xor crc(14);
  result(15) := d(7) xor d(3) xor crc(7) xor crc(11) xor crc(15);

  return result;
end crc_ccitt_step;
```

Hardware Realization as a USR-Instruction

VHDL

```
 CASE  i_type  IS
...
 WHEN OTHERS => -- op_USR
       CASE i_USR IS
                WHEN "00110" =>  -- USR 6
                   -- push_stack( "00000000000000000000111" );
                   pop_stack;
                   -- tos_new <= multiply(tos_reg, nos_reg);
                   tos_new <= crc_ccitt_step(tos_reg, nos_reg);
                WHEN OTHERS =>
                  IF  inst=INT_OP  THEN
                     push_stack(tos_reg);
                     tos_new <= status;
                  END IF;
               END CASE;
  END CASE;
```

- Data stack effekt
- Return stack effekt
- Sequencer

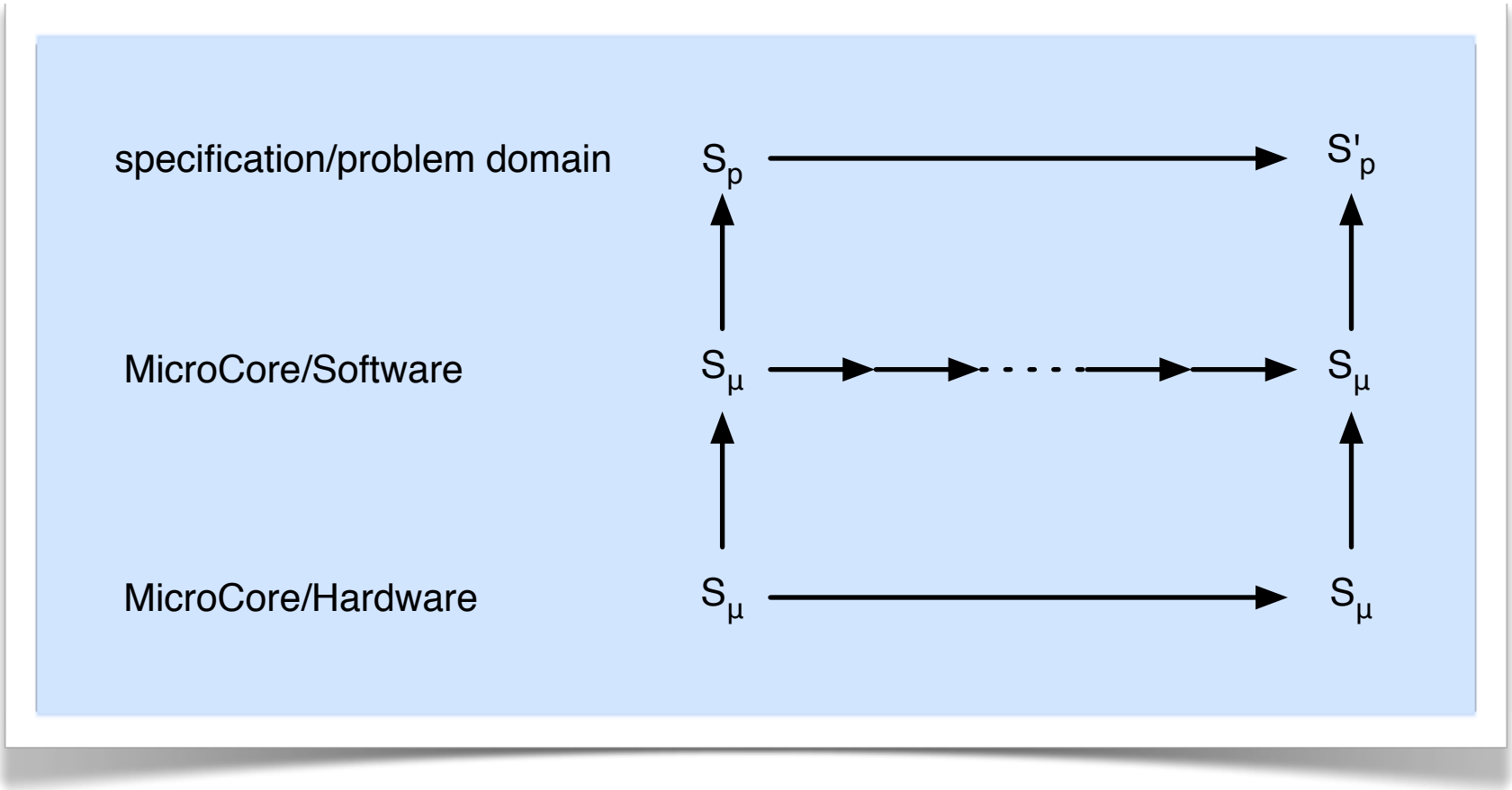
Forth

```
5 USR: crc-step ;USR

: crc-ccitt ( addr len -- crc )
  $FFFF rot rot
  bounds ?DO  I @ crc-step
              LOOP
;
```

Proof obligations

- In what areas do you need to think about the problem?



Future

- Other candidates for realization in hardware:
  - Multiplication
  - MD5
  - RSA
  - digital filters
  - FFT

- **MicroCore**
  - Experiments in computer architecture
  - Overflow detection already present
  - Safe program execution
  - Tagged memory for type information