TCE Tour: Screencast

- These are accompanying slides for a set of TCE screencast clips available at http://tce.cs.tut.fi/index.php/home/screencasts
- The set of videos goes through the most important tools in TCE by means of a simple CRC example application
- Starts from C, ends with a TTA+program running on an FPGA board
Intro and Exploration (about 7 minutes)

- Start with the application code in C
- minimal.adf is shipped in TCE:
  - a minimal set of resources in TTAs supported by the tcecc compiler
- reflect.vhdl is a VHDL implementation of a custom operation which will be added later to the design
Intro and Exploration

- Open the minimal.adf to the Processor Design GUI
  - “prode minimal.adf &”
Intro and Exploration

- The main.c has a simple string “TCE rocks!” for which we are going to compute the CRC
Notes: To test a different CRC standard, modify crc.h.

Notes: Only crc-32 is used.

#ifndef _DEBUG
#include <stdio.h>
#endif
#ifndef _DEBUG
#include "crc.h"
#endif

volatile crc result = 0;
unsigned char test[3] = {'A', 'B', 'C'};

#define LENGTH 10

int main(void) {
    /* Compute the CRC of the test message, more efficiently. */
    result = crcFast(test, LENGTH);

    ifdef _DEBUG
    printf("The crcFast() of \"ABC\" is 0x%04x", result);
    endif _DEBUG

    return 0;
}
Function: crcFast()
Description: Compute the CRC of a given message.
Notes: CRCInit() must be called first.
Returns: The CRC of the message.

```c
unsigned char* crcFast(unsigned char* const message[], int nBytes)
{
    unsigned char* data;
    unsigned char byte;

    /* Divide the message by the polynomial, byte at a time. */
    for (byte = 0; byte < nBytes; ++byte)
    {
        data = REFLECT_Init(message[byte]) ^ (remainder >>> (WIDTH - 8));
        remainder = crcTable[data] ^ (remainder << 8);
    }

    /* The final remainder is the CRC. */
    return (REFLECT_Remainder(remainder) ^ FINAL_XOR_VALUE);
}
```
Let's see how well the CRC code runs with the smallest supported TTA

- Compile the code to the TTA with the retargetable tcecc compiler
- Load the processor architecture description and the compiled program to the Processor Simulator GUI (Proxim)
Intro and Exploration

- Proxim's main window displays the disassembly of the TTA program
- The minimal.adf has only one bus, thus the moves cannot be parallelized
Proxim's machine window:

- Visualizes the TTA processor when running the given program
- Single stepping the assembly code highlights the transport paths in the processor accessed by the moves in the current instruction
- Allows inspecting the values in programmer-visible registers of the TTA such as FU ports, the utilization of the components (color coding), etc.
- Most importantly for this case, the simulator displays the total cycle count 6109 (number of TTA instructions executed)
- Also statistics for the different operations executed, registers used etc. can be produced to guide manual exploration of the architecture
  - “info proc stats”
The minimal.adf has only 5 registers, the CRC algorithm can use more as we saw from the stats.

Let's add some more registers using the Processor Designer (ProDe)
  - In this case we double the number to 10 registers.

Recompile the program for the new architecture with 10 registers using tcecc.

This time we'll load the processor+program to the command line interface of the simulator (ttasim)
  - "info proc cycles" works here also and produces the cycle count 3116 which is almost halved from the one we got using a machine with only 5 registers.
```
[1] 8294
otksi@untamo:~/intro$ prode minimal.adf &
[2] 8516
otksi@untamo:~/intro$ tcecc -O2 -a minimal.adf -o crc.tpef crc.c main.c
otksi@untamo:~/intro$ proxim minimal.adf crc.tpef &
[2]+ Done
otksi@untamo:~/intro$ ttasim -a minimal.adf -p crc.tpef

(ttasim) info proc cycles
3116
(ttasim) 
```
Intro and Exploration

- Next we'll try the Design Space Explorer tool
- The tool is used to launch “explorer plugins” which perform modifications to the target and measure their effect to:
  - cycle count
  - area estimate
  - energy estimate
  - longest path delay estimate
- The plugins can be fully automated or semi-automated
  - Can implement a loop that explores multiple points in the design space or just generate one new design space point (processor configuration)
- “explore -g” prints a list of available exploration plugins
  - In this example we use the GrowMachine plugin which adds basic resources to the machine until the cycle count does not drop anymore significantly
The explorer is used by first adding the software of the application to a Design Space Database (dsdb).

Then we launch the explorer plugin which produces one or more new “configurations” to the DSDB along with their characteristics data (at least cycle counts).

- Plugins usually have parameters which can also be configured through explorer.

In this case 2 new configurations were produced after starting the GrowMachine from our minimal.adf starting point.

The best cycle count we got using this explorer plugin is 690.

Let's see with ProDe how the generated best architecture looks like.
As we can see, the GrowMachine plugin has added more buses and FUs to the machine
- Currently a brute-force approach of incrementing the current resource set with a constant factor is used
For example, the machine has 9 buses (instead of 1), many more function units and additional two register files
Profiling and Using a Custom Operation (about 4 minutes)

- The GrowMachine plugin managed to squeeze the cycle count down to **690** by just duplicating resources.
- We are not happy with this number yet as we know it can get much lower when some custom hardware is used.
- This video shows how to profile the application and use a custom operation (special function unit) to accelerate a “hot spot” in the CRC program.
First we'll compile the program with procedure inlining disabled so we get a proper function profile of the program.
Simulate the program:

- Note that the cycle count has increased due to the disabled inlining to 4917
- Verify the program by dumping the computed CRC number from memory
Profiling and Using a Custom Operation

Program profile:
- To find out the “hot spot” in the program, we highlight the top executed instructions.
- We find out the instructions in the _reflect() function are executed very frequently, thus it's a potential candidate for acceleration with a custom operation (special function unit).
We find out that the reflect function is called through macros REFLECT_DATA and REFLECT_REMAINDER in the core loop of the C code.

The reflect() computes a “bit reflection”:
- Reverses bits like a mirror was placed in the middle of the word.
- We see from the macros that it’s done only for word sizes 8 and 32 bits.
```c
  /* oTable[dividend] = remainders; 
 */
/* oInit() */

// Function: oFast()
// Description: Compute the CRC of a given message.
// Notes: oInit() must be called first.
// Returns: The CRC of the message.

// oFast(unsigned char const message[], int nBytes)
  oFast(unsigned char const message[], int nBytes)
  {
    unsigned char rem;
    int byte;

    // Divide the message by the polynomial, a byte at a time.
    for (byte = 0; byte < nBytes; ++byte)
    {
      data = REFLECT_INIT(message[byte]) ^ (rem >> (WIDTH - 8));
      rem = oTable[rem] ^ (rem << 8);
    }

    // The final remainder is the CRC.
    return (REFLECT_REMINDER(rem) ^ FINAL_KER_VALUE);
  }
/* oFast() */
```
```
#define REFLECT_DATA(X)  (*((unsigned char *)(X), 8))
#define REFLECT_REMINDER(X) (X) & 0x000000FF

static unsigned long reflect(unsigned long data, unsigned char nBits)
{
    unsigned long reflection = 0x00000000;
    unsigned char bit;
    for (bit = 0; bit < nBits; ++bit)
    {
        * If the LSB bit is set, set the reflection of it.
        if ((data & 0x01) == 1)
        {
            reflection |= (1 << ((nBits - 1) - bit));
        }
        data = (data >> 1);
    }
    return (reflection);
}
```
Profiling and Using a Custom Operation

- The reflect() function is extremely simple and efficient to implement in hardware (just wiring and shifting if necessary), but looks like a heavy loop when implemented in C code.
- Let's create a custom operation for the REFLECT:
  - Custom operations added to TCE using a tool called Operation Set Editor (OSEd).
- First we add general “static” information about the operation like its name and the number and type of inputs and outputs.
Profiling and Using a Custom Operation

- The let's add a simulation behavior description for the operation
- We can copy the original C code to the simulation behavior definition, just define:
  - Reads from operation inputs (UINT(1), UINT(2)) to variables in the C code
  - Write result to the operation output (IO(3))
- The simulation behavior is loaded runtime to the processor simulator
  - It's a “plugin” module which needs to be compiled
  - Build it with OSEd
  - Test that the simulation behavior definition works using the operation behavior simulator
static unsigned long
reflect(unsigned long data, unsigned char nBits)
{
  unsigned long reflection = 0x00000000;
  unsigned char bit;

  /* Reflect the data about the center bit. */
  for (bit = 0; bit < nBits; ++bit)
  {
    /* If the LSB bit is set, set the reflection of it. */
    if ((data & 0x01))
      reflection = (1 << ((nBits - 1) - bit));
    data = (data >> 1);
  }

  return (reflection);
}

/* reflect() */

/* Function: crcSlow() */
/* Description: Compute the CRC of a given message. */
/* Notes: */
/* Returns: The CRC of the message. */
Adding SFU to the Machine and Using it in C Code (1.5 minutes)

- Now that we have defined a new custom operation to the TCE, we can use it in our TTA in a special function unit and execute it from our C code
- Add the custom operation to a new function unit in the TTA with the Processor Designer tool
  - Add a function unit
  - Add ports to the function unit
  - Add the operation to the function unit
  - Edit the operations port bindings, pipeline resource usage, and latency
- In this case we are certain that the REFLECT operation can be done in 2 cycles in hardware
  - Probably 1 cycle would be enough due to the operation's simplicity, but we “play it safe”
Adding SFU to the Machine and Using it in C Code (1.5 minutes)

- Now the architecture supports the REFLECT custom operation with the added function unit.
- Let's now use the REFLECT operation from our C code to accelerate the algorithm.
- First add:
  - `#include "tceops.h"
  - This brings in the macros that are used to invoke TTA operations manually.
- Then call the REFLECT operation through a TCE operation macro:
  - `_TCE_REFLECT(...);`
Filename: crc.c

Description: Slow and fast implementations of the CRC standards.

Notes: The parameters for each supported CRC standard are defined in the header file crc.h. The implementations here should stand us to further additions to that list.

* Copyright (c) 2000 by Michael Barr. This software is placed into the public domain and may be used for any purpose. However, this notice must not be changed or removed and no warranty is either expressed or implied by its publication or distribution.

* TIE version modified by Otto Esko

* Changes:
  * Only CRC-32 version is used, other versions are omitted.
  * crc32x version is preserved although it's not used.

#include "crc.h"
#include "toplevel.h"

/* Derive parameters from the standard-specific parameters in crc.h. */
#define WIDTH 32
#define TOPBIT (1 << (WIDTH - 1))
#define REFLECT.Uint(x) ((unsigned char) reflect(x, 8))
#define REFLECT.Reminder(x) ((crc) reflect(x, WIDTH))

/* Function reflect() */
/* Description: Reorder the bits of a binary sequence, by reflecting then about the middle position. */
/* Notes: No checking is done that nbits <= 32. */

FU: rotator
  (rotl, rotr)

FU: reflect
  (reflect)

RF: 10x32

RF: bool
  2x1
```c
void crcFast(unsigned char const *message, int nBytes)
{
    unsigned char remainder = INITIAL_REMINDER;
    unsigned char data;
    int byte;

    // Divide the message by the polynomial, a byte at a time.
    for (byte = 0; byte < nBytes; ++byte)
    {
        data = REFLECT_DIN(message[byte]) ^ (remainder >> (WIDTH - 8));
        remainder = crcTable[byte] ^ (remainder << 8);
    }

    // The final remainder is the CRC.
    return (REFLECT_REMINDER(remainder) ^ FINAL_XOR_VALUE);
}
/* crcFast */
```
Description: Compute the CRC of a given message.

Notes: crcInit() must be called first.

Returns: The CRC of the message.

```c
unsigned char const message[], int nBytes;
unsigned char data;
unsigned char output;
unsigned char remainder = INITIAL_REMAINER;
for (byte = 0; byte < nBytes; byte++)
    data = message[byte];
    crc = crcReflect(data); (WITH 16)
    remainder = crcReflect(remainder) (WITH 8);
    output = output ^ (remainder >> 8);
    remainder = remainder << 8;
    crc = crcReflect(remainder); (WITH 16)
    output = output ^ (remainder >> 8);
}
```

The final remainder is the CRC.
Adding SFU to the Machine and Using it in C Code (1.5 minutes)

- Finally, recompile the code which now uses the custom operation, verify that the program still works correctly, and see its effect to the cycle count using the simulator
  - Cycle count now dropped to 403
  - By using custom operation we reached a lower cycle count with much less hardware

- Now we could use explorer to increase the performance
  - Current architecture has only one bus
  - By increasing concurrency we would reach lower cycle count
Adding Implementation of the SFU to the Hardware Database (50 sec)

- Now we have found a good custom operation to accelerate our algorithm and used it in our architecture and C code
- In order to generate VHDL for the processor, we still need to add an implementation of the SFU to a Hardware Database (HDB)
- Of course, implementing the SFU might take a bit longer than the 50 sec, thus we use a previously implemented VHDL block for demonstration purposes :) 
- HDBEditor is a GUI for editing HDBs, we use it to add the implementation to an HDB along with the data needed to generate a processor
  - The names of the input/output ports and the entity name in the VHDL, etc..
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity reflect is
  generic (
    dataw : integer := 32);
  port (
    data_in : in std_logic_vector(dataw-1 downto 0);
    size : in std_logic_vector(4 downto 0);
    reflect_data : out std_logic_vector(dataw-1 downto 0);
  );
end reflect;

architecture reflect of reflect is
begin
  signal reflect_temp : std_logic_vector(dataw-1 downto 0);
  -- reflect
  process (data_in, size)
  begin
    variable i : integer range 0 to dataw-1 := 0;
    process reflect_process
    begin
      case size is
        when '100100' =>
          i := 0;
          for i in 0 to dataw-1 loop
            reflect_temp(i) <= data_in(dataw-1-i);
          end loop; -- i
        when others =>
          if dataw > 8 then
            i := 0;
            for i in 8 to dataw-1 loop
              reflect_temp(i) <= '0';
            end loop; -- i
          elseif dataw = 8 then
            i := 0;
            for i in 0 to 7 loop
              reflect_temp(i) <= data_in(7-i);
            end loop; -- i
          else
            i := 0;
            for i in 0 to dataw-1 loop
              reflect_temp(i) <= data_in(dataw-1-i);
            end loop; -- i
          end if;
      end case;
    end process reflect_process;
  end process;
end reflect;
Adding Implementation of the SFU to the Hardware Database (50 sec)

- Now we have added an implementation of the REFLECT SFU to a HDB.
- Finally, we need to connect the architecture of the FU in our TTA architecture file to this implementation.
  - Use automated exploration plugin for this.
- In TCE, architecture of the processor components and the actual implementation are separated.
  - Architecture components (in ADF files edited with ProDe) are connected to HDB implementations through an Implementation Definition File (IDF).
  - Architecture definition file (ADF), implementation definition file (IDF) and one or more Hardware Databases (HDB) form a “processor configuration” that can be outputted as a VHDL implementation.
Generating the Processor (32 sec)

- Now we have all we need to generate the processor implementation in VHDL
- For this we use the Processor Generator (ProGe) tool which can be invoked from the command line or from the ProDe GUI
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.globals.all;
use work.util.all;
use work.imem_all;
use work.toplevel_params.all;

entity toplevel is

port (
    clk : in std_logic;
    rstx : in std_logic;
    busy : in std_logic;
    imem_en_x : out std_logic;
    mem_addr : out std_logic_vector(IMEMADDR-1 downto 0);
    mem_data : in std_logic_vector(IMEMWIDTH-1 downto 0);
    fuletcher_in : in std_logic_vector(IMEMWIDTH-1 downto 0);
    fuletcher_out : out std_logic_vector(fuletcher.data-1 downto 0);
    fuletcher_addr : out std_logic_vector(fuletcher.addr-1 downto 0);
    fuletcher_mem_en_x : out std_logic_vector(0 downto 0);
    fuletcher_mem_mask_x : out std_logic_vector(0 downto 0));

end toplevel;

architecture structural of toplevel is

signal inst_fetch_ra_out_wire : std_logic_vector(IMEMADDR-1 downto 0);
signal inst_fetch_ra_in_wire : std_logic_vector(IMEMADDR-1 downto 0);
signal inst_fetch_pc_in_wire : std_logic_vector(IMEMADDR-1 downto 0);
signal inst_fetch_pc_load_wire : std_logic_vector(IMEMADDR-1 downto 0);
signal inst_fetch_pc_opcode_wire : std_logic_vector(0 downto 0);
signal inst_fetch_fetch_en_wire : std_logic;
signal inst_fetch_fetchblock_wire : std_logic_vector(IMEMADDR-1 downto 0);
signal fuletcher_tidata_wire : std_logic_vector(29 downto 0);
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tidata_wire : std_logic_vector(31 downto 0);
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
signal fuletcher_tload_wire : std_logic;
Change Load-store Unit to an Avalon Bus

Load/store Unit (36 secs)

- Next we'll use an FPGA board to test the processor
- For this we need to change the load-store unit function unit implementation to one that supports Altera's Avalon interface
  - We'll use the Altera Memory Mapped Interface
  - TTA acts as a master on the bus
  - This way we can use Altera's IP-components
- This can be done quickly with the Processor Designer tool
Using Avalon LCD for Output (21 secs)

- Now the TTA can interface with the memory (and other I/O) in the FPGA board using the Avalon bus
- Finally, we need a device to produce some output from our CRC computation to verify it actually works
- For this we use an LCD screen connected through the Avalon bus
  - We use the LCD controller from SOPC Builder's IP component library
- The LCD controller is connected to the Avalon Memory Mapped bus interface so we can define a putchar() function (which is used by printf()) that writes characters to the controller's memory mapped registers
Generate the Bit Image of the Program Memory and Synthesize the Design (2 minutes)

Finally, to get the TTA running on the FPGA we need to generate a bit image of the program memory
  - Use command line tool generatebits
Load the VHDL files of the generated TTA processor to the Altera's Quartus II tool to synthesize the design to the FPGA

- We add a layer on top of the ProGe generated toplevel.vhdl (not displayed in the video)
  - In this case the instruction memory is very small so we implement as “logic"
  - Synthesize tools optimizes it into a small space of internal memory banks and logic
- The external interface of the new layer is the external buses of the load-store unit we added, and control signals (clk, reset)
  - The LSU interface is actually the Avalon interface
- Then in Altera's SOPC builder we export TTA as a component to the design along with the onchip memory and the LCD component and connect them all to the Avalon bus
  - TTA is the Avalon Master and the memory and LCD controller are slaves
- Synthesize the design to the FPGA and note how many of the logic elements were consumed of the FPGA by our TTA
- Finally, upload the design to the FPGA board
Quartus II
Version 8.0

Download New Software Release

Documentation
use work.tta_pal.vhd;
use work.tta04_pal.vhd;

entity tta_processor is
  port(
    -- CLOCK_50 : in std_logic;
    -- KEY : in std_logic_vector(3 downto 0);
    clk : in std_logic;
    rst_n : in std_logic;
    address : in std_logic_vector(11-1 downto 0);
    dbyteenable : out std_logic_vector(11 downto 0);
    dready : out std_logic;
    dwrite : out std_logic;
    dworddata : out std_logic_vector(32-1 downto 0);
    dreq : out std_logic;
    dwe : out std_logic;
    dweenable : out std_logic_vector(32-1 downto 0);
    dwaitrequest : out std_logic;
  );

end tta_processor;

architecture top of tta_processor is

component toplevel is
  port(
    clk : in std_logic;
    rdy : in std_logic;
    busy : in std_logic;
    mem_en_x : out std_logic;
    mem_en_y : out std_logic;
    mem_data : in std_logic_vector(MEMORY_WIDTH-1 downto 0);
    mem_addr : in std_logic_vector(MEMORY_WIDTH-1 downto 0);
    ps_init : in std_logic_vector(MEMORY_WIDTH-1 downto 0);
    full_a_address : out std_logic_vector(full_a_address_depth-1 downto 0);
    full_a_bytenable : out std_logic_vector(full_a_bytenable_depth-1 downto 0);
    full_a_read : out std_logic_vector(0 downto 0);
    full_a_write : out std_logic_vector(0 downto 0);
    full_a_worddata : out std_logic_vector(full_a_worddata_width-1 downto 0);
    full_a_wordwrite : out std_logic;
    full_a_wrequest : out std_logic;
    full_b_address : out std_logic_vector(full_b_address_depth-1 downto 0);
    full_b_bytenable : out std_logic_vector(full_b_bytenable_depth-1 downto 0);
    full_b_read : out std_logic_vector(0 downto 0);
    full_b_write : out std_logic_vector(0 downto 0);
    full_b_worddata : out std_logic_vector(full_b_worddata_width-1 downto 0);
    full_b_wordwrite : out std_logic;
    full_b_wrequest : out std_logic;
  );

end component;

signal dummy_wire : std_logic;
signal clk_wire : std_logic;
signal reset_n_wire : std_logic;
signal busy_wire : std_logic;
signal mem_en_x_wire : std_logic_vector(MEMORY_WIDTH-1 downto 0);
signal mem_en_y_wire : std_logic_vector(MEMORY_WIDTH-1 downto 0);
signal ps_init_wire : std_logic_vector(MEMORY_WIDTH-1 downto 0);
signal mem_data_wire : std_logic_vector(MEMORY_WIDTH-1 downto 0);
signal full_a_address_wire : out std_logic_vector(full_a_address_depth-1 downto 0);
signal full_a_bytenable_wire : out std_logic_vector(full_a_bytenable_depth-1 downto 0);
signal full_a_read_wire : out std_logic_vector(0 downto 0);
signal full_a_write_wire : out std_logic_vector(0 downto 0);
signal full_a_worddata_wire : out std_logic_vector(full_a_worddata_width-1 downto 0);
signal full_a_wordwrite_wire : out std_logic;
signal full_a_wrequest_wire : out std_logic;
signal full_b_address_wire : out std_logic_vector(full_b_address_depth-1 downto 0);
signal full_b_bytenable_wire : out std_logic_vector(full_b_bytenable_depth-1 downto 0);
signal full_b_read_wire : out std_logic_vector(0 downto 0);
signal full_b_write_wire : out std_logic_vector(0 downto 0);
signal full_b_worddata_wire : out std_logic_vector(full_b_worddata_width-1 downto 0);
signal full_b_wordwrite_wire : out std_logic;
signal full_b_wrequest_wire : out std_logic;

component inst_mcm_logic is
  generic(
    addr : in std_logic;
    instr : in std_logic;
    addrv : in std_logic_vector(ADDR_WIDTH-1 downto 0);
    instrv : in std_logic_vector(INSTR_WIDTH-1 downto 0);
  )
  port (clk : in std_logic;
    addr : in std_logic_vector(addrv-1 downto 0);
    instr : in std_logic_vector(instrv-1 downto 0));

end component;

Info: Your system is ready to generate.

For Help, press F1
The End – Thanks for Your Attention!