TOWARDS SOFTWARE-DEFINED SILICON: APPLYING LLVM TO SIMPLIFYING SOFTWARE

TEEMU RINTA-AHO (TEEMU.RINTA-AHO@ERICSSON.COM),
PEKKA NIKANDER, ADNAN GHANI, SAMEER D. SAHASRABUDDHE
NOMADICLAB, ERICSSON RESEARCH FINLAND

WISH-3, CHAMONIX, FRANCE
APRIL 2, 2011

PRESENTATION OUTLINE

- Introduction
- Background
  - Click Modular Router
  - Stanford NetFPGA prototyping board
  - High Level Synthesis (HLS)
  - AHIR
- Overall Approach
- Use Case
- Conclusions

INTRODUCTION

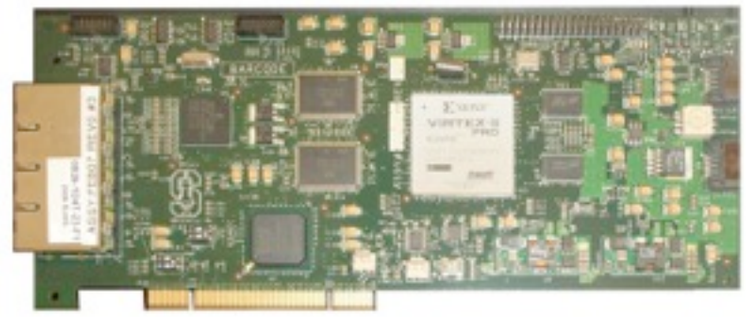
- The starting point: study the applicability of HLS to packet networking
- The practical exercise: implement a Click-to-NetFPGA toolchain
- The motivation: study the applicability of modular compiler optimisations for atypical/generated hardware
  - CPUs and ASIC/FPGA seen as two ends of a design space
  - What could be done in the middle ground?
- This work started only a year ago; this presentation is to publish intermediate results on a work-in-progress project and hopefully get others interested
  - We are happy to share the source code

CLICK MODULAR ROUTER

- Software platform (C++) for building different kinds of packet-processing nodes / functions, or “routers”
- Defines a software router from packet processing elements
  - Open source, by Eddie Kohler (MIT)
- Runs in Linux/BSD/Darwin userspace & Linux kernel
  - (BTW, we are also working on a FreeBSD kernel version)
- 100+ existing elements, e.g. ARPResponder, Classifier
- Easy to add new elements
- Easy to build a custom “routers”

STANFORD NETFPGA

- A PCI network interface card with an FPGA
  - 4 x 1G Ethernet interface
- Line-rate, flexible, and open platform
- For research and classrooms



- More than 1,000 NetFPGA systems deployed
- A few open-source, Verilog-based reference designs
- A newer, NetFPGA 10G card is also available
  - 4 x 10G Ethernet interface, bigger FPGA, faster PCI interface

HIGH LEVEL SYNTHESIS (HLS)

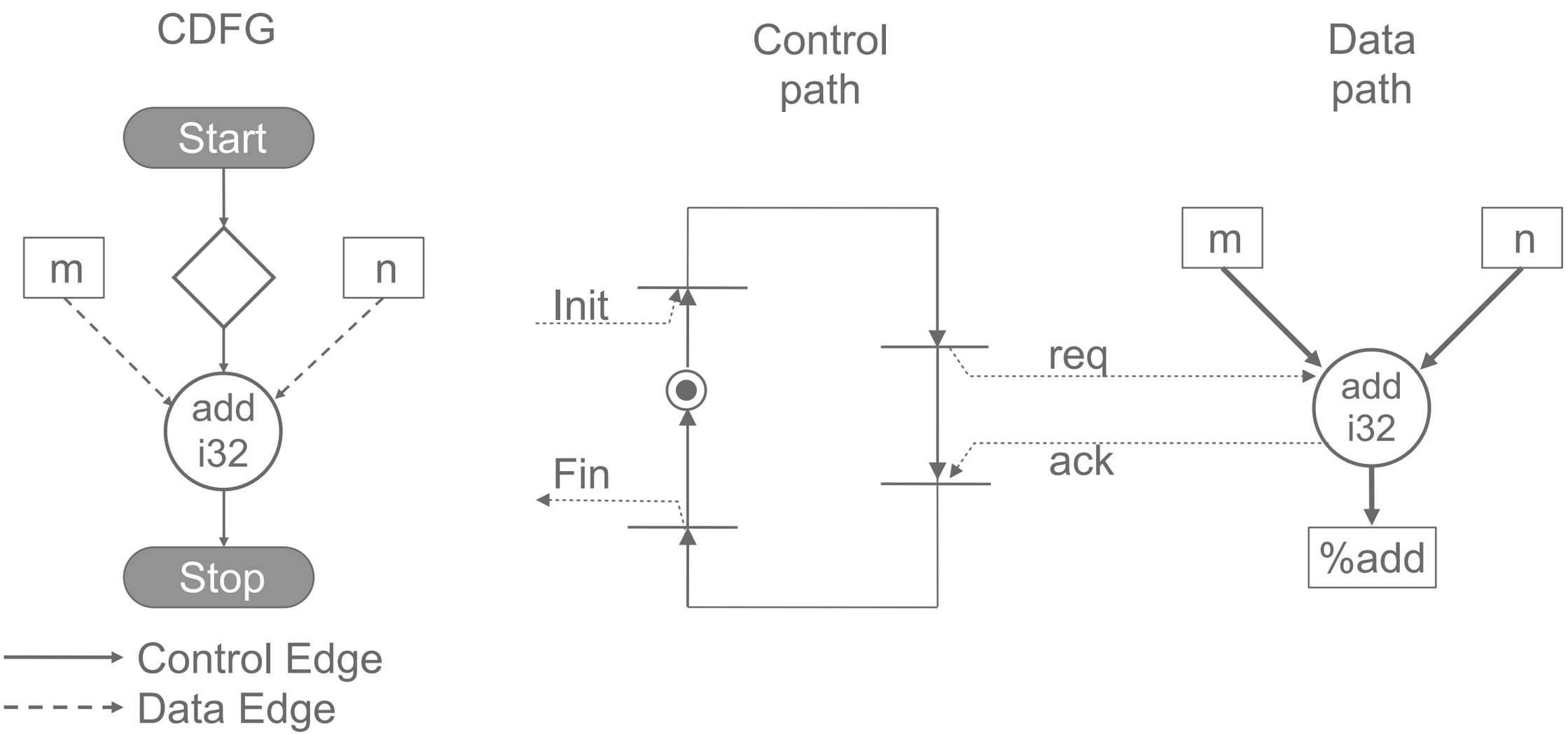
- A high level program → hardware (RTL)
  - Typically: C/C++/System C → Verilog/VHDL
- Currently available tools' limitations and "features"
  - Designed for hardware professionals; for "writing hardware" in C
  - Support only a subset of C/C++, excluding e.g.:
    - dynamic memory
    - function pointers and virtual methods
    - recursion
  - Cannot handle existing software-oriented code
    - e.g. Click elements written using full C++
  - Closed-source commercial products

AHIR
"A HARDWARE INTERMEDIATE REPRESENTATION"

- LLVM backend for generating VHDL
  - To-be open source, by IIT Bombay (India)
  - Factorises the system into control, data, and storage
    - Supports scalable optimisations and analyses
  - Current limitations: no recursion or function pointers, otherwise full C
  - Generates a VHDL module out of each LLVM IR function
- Design = Set of modules with I/O channels
  - I/O through a simple VHDL "library", resembling Unix pipes

AN AHIR EXAMPLE
CONVERTING AN LLVM INSTRUCTION TO VHDL

- Simple Addition Example
- C code: **`int d = m + n;`**
- Equivalent SSA: **`%d = add i32 %m, %n`**



PRESENTATION OUTLINE

- Introduction
- Background
  - Click Modular Router
  - Stanford NetFPGA prototyping board
  - High Level Synthesis (HLS)
  - AHIR
- Overall Approach
- Use Case
- Conclusions

OVERALL APPROACH

- Click C++ → “hardware friendly” LLVM IR → VHDL
  - New Click “target”, similar to BSD/Linux specific differences
    - E.g. use table-based memory allocation instead of skb/heap-based
- Click2LLVM: Click configuration → LLVM IR
  - Use Click library functions to set up a “Click router” in memory
  - Dump the memory as a set of constants, expressed in LLVM IR
- LLVM IR → “hardware friendly” LLVM IR
  - Fixing “this” arguments to constants
  - Aggressive constant propagation, aggressive inlining
  - Together resulting in e.g. removing virtual function calls
- AHIR: “hardware friendly” LLVM IR → VHDL

Click NetFPGA target library

Click element source

llvm-g++

Click element IR

llvm-g++

Click element object

llvm-link

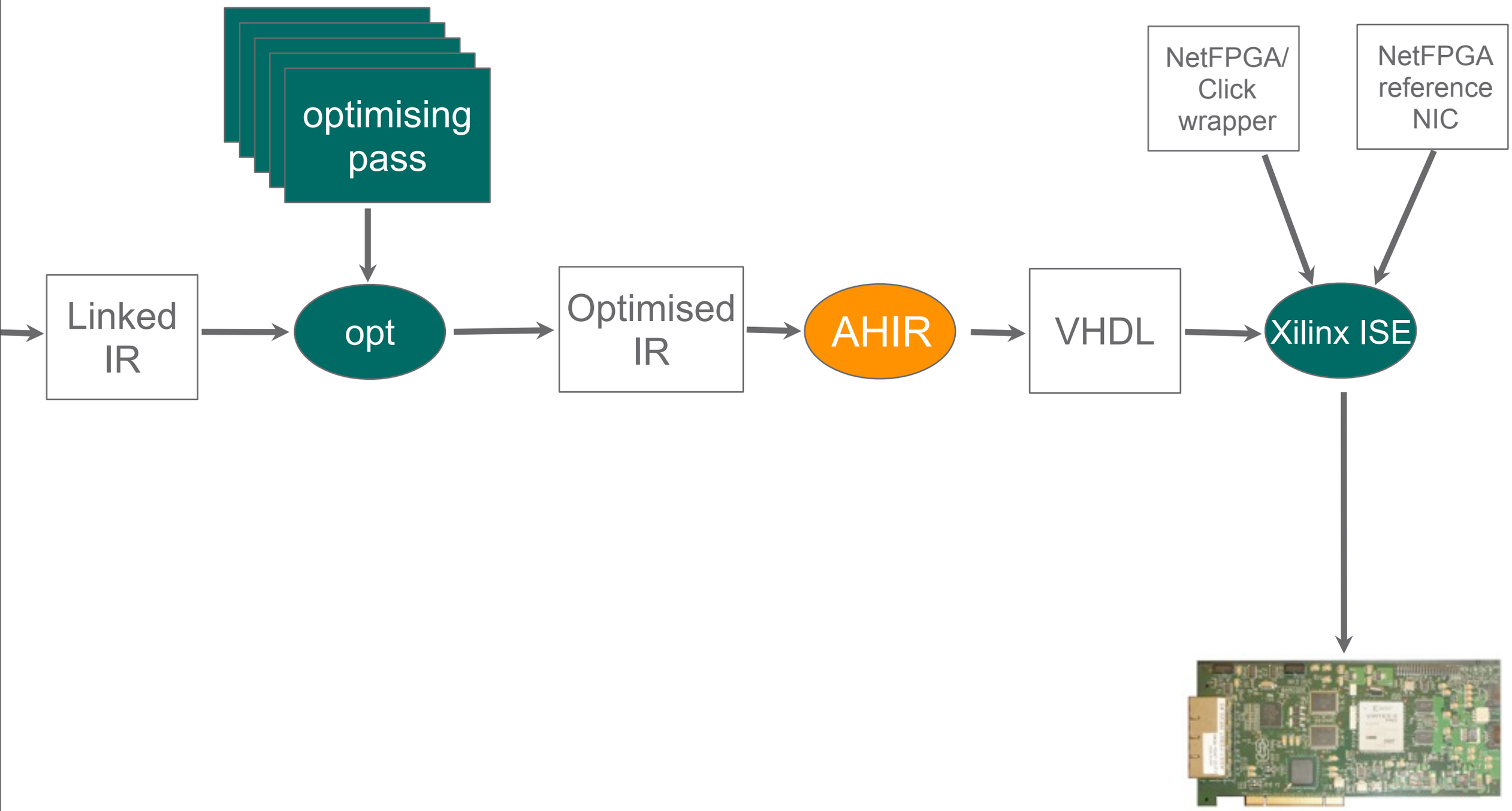
Linked IR

Click host target library

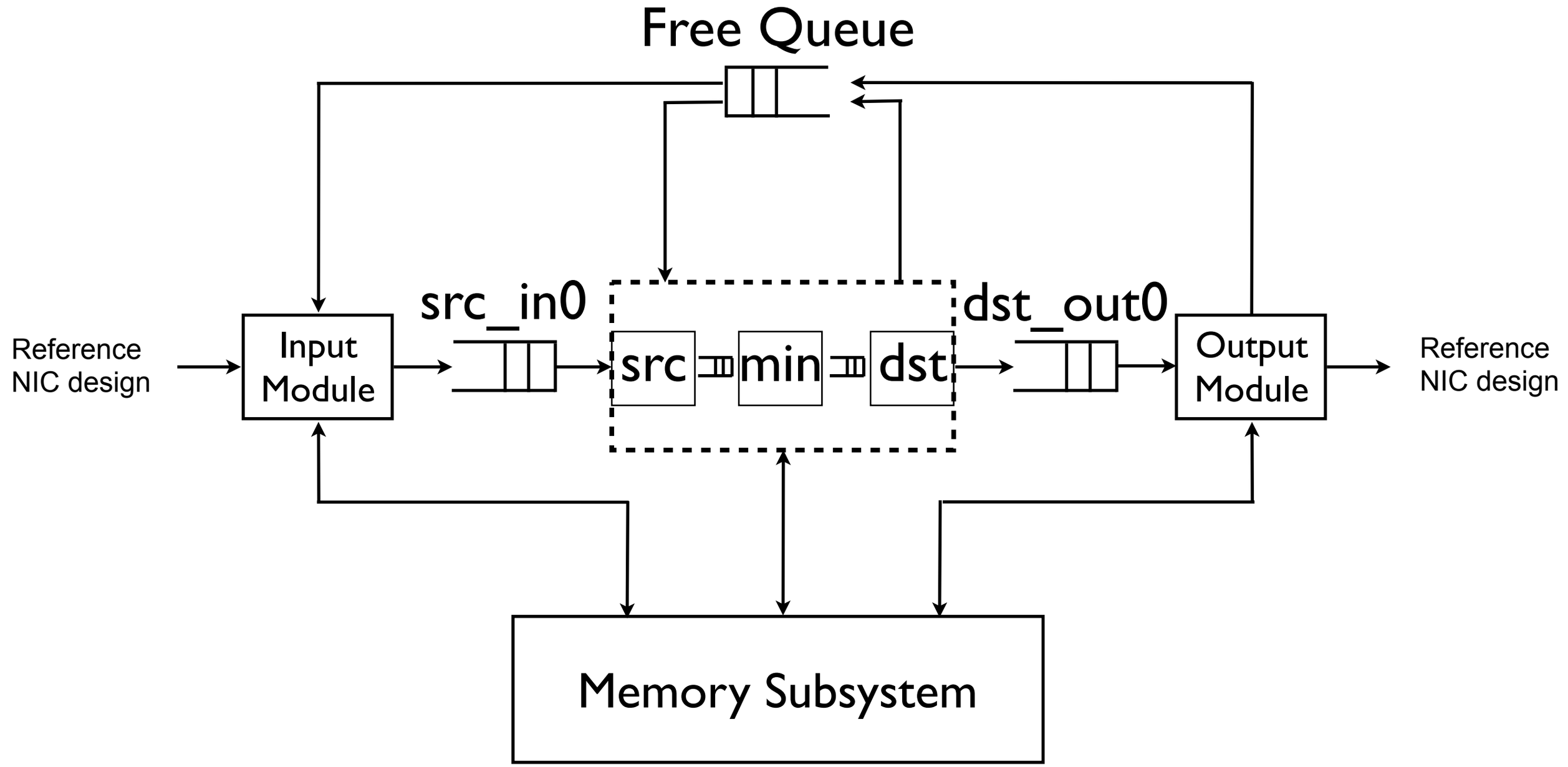
Click2-LLVM

Click config IR

Click config source


optimising pass
Linked IR
opt
Optimised IR
AHIR
VHDL
NetFPGA/ Click wrapper
NetFPGA reference NIC
Xilinx ISE

CLICK/NETFPGA WRAPPER



NETFPGA WRAPPER FOR THE CLICK/AHIR SYSTEM

ERICSSON

- NetFPGA processes packets on the fly, word by word
- Click only transfers pointers to Packets
  - Packets stay in the same memory location
- The wrapper receives and sends NetFPGA words
  - Stores them locally as a Packet in a memory subsystem
- Memory is managed as a queue of free locations
  - InputModule reads a free location from the "free queue", then receives data from the NetFPGA interface & stores it
  - OutputModule writes the data on the NetFPGA interface, then returns the pointer to the free queue
- HLS includes elements from Click configuration
  - Click elements may create or destroy packets using direct access to the free queue

PRESENTATION OUTLINE

- Introduction
- Background
  - Click Modular Router
  - Stanford NetFPGA prototyping board
  - High Level Synthesis (HLS)
  - AHIR
- Overall Approach
- Use Case
- Conclusions

USE CASE: CLICK.CONF

```
require(package "minimal-package");

src :: FromFPGA;
min :: Minimal;
dst :: ToFPGA;

src -> min -> dst;
```

USE CASE: CLICK.CONF

```
require(package "minimal-package");

src :: FromFPGA;
min :: Minimal;
dst :: ToFPGA;

src -> min -> dst;
```

Dynamically link external Click elements

USE CASE: CLICK.CONF

```
require(package "minimal-package");

src :: FromFPGA;
min :: Minimal;
dst :: ToFPGA;

src -> min -> dst;
```

Dynamically link external Click elements

Declare and name the used elements

USE CASE: CLICK.CONF

```
require(package "minimal-package");

src :: FromFPGA;
min :: Minimal;
dst :: ToFPGA;

src -> min -> dst;
```

Dynamically link external Click elements

Declare and name the used elements

Define packet processing order

USE CASE: SIMPLE_ACTION()

```
Packet *
Minimal::simple_action(Packet *p) {
    unsigned char *data = p->uniqueify()->data();
    data[0] ^= 0x0F;
    return p;
}
```

The packet processing part of a valid, yet an extremely simple, Click element named “Minimal”.

Written in C++ without any special target-specific requirements or restrictions.

USE CASE: AHIR_GLUE_MIN()

```
@1 = constant [8 x i8] c"min_in0\00"
@GV_3 = constant [1 x %"struct.Element::Port"] [%"struct.Element::Port" {
  [16 x i8] c"dst_in0\00\00\00\00\00\00\00\00", %struct.Element* undef, i32 undef }]
@min = constant %struct.Minimal { ..., @GV_3, ... }   ; Struct contents not fully shown

define void @ahir_glue_min() {
  %0 = call i32 @read_uintptr(i8* getelementptr inbounds ([8 x i8]* @1, i32 0, i32 0))
  %1 = inttoptr i32 %0 to %struct.Packet*
  call void @element_push(%struct.Element* getelementptr inbounds
    (%struct.Minimal* @min, i32 0, i32 0), i32 0, %struct.Packet* %1)
  ret void
}
```

The new, generated, wrapper function together with generated global constants, in LLVM IR form.

USE CASE: AHIR_GLUE_MIN() OPTIMISED

```
@1 = internal constant [8 x i8] c"min_in0\00"
@GV_3 = constant [1 x %"struct.Element::Port"] [%"struct.Element::Port" {
  [16 x i8] c"dst_in0\00\00\00\00\00\00\00\00", %struct.Element* undef, i32 undef }]

define void @ahir_glue_min() ssp {
  %tmp = tail call i32 @read_uintptr(i8* getelementptr inbounds ([8 x i8]* @1, i32 0, i32
0))
  %tmp3 = inttoptr i32 %tmp to %struct.Packet*
  %tmp4 = getelementptr inbounds %struct.Packet* %tmp3, i32 0, i32 3
  %tmp5 = load i8** %tmp4, align 4
  %tmp6 = load i8* %tmp5, align 1
  %tmp7 = xor i8 %tmp6, 15
  store i8 %tmp7, i8* %tmp5, align 1
  %tmp8 = icmp eq i32 %tmp, 0
  br i1 %tmp8, label %_ZN7Element4pushEiP6Packet.exit, label %bb.i

bb.i:
  tail call void @write_uintptr(i8* getelementptr inbounds (
    [1 x %"struct.Element::Port"]* @GV_3, i32 0, i32 0, i32 0, i32 0), i32 %tmp)
  ret void

_ZN7Element4pushEiP6Packet.exit:
  ret void
}
```

The optimised wrapper function. Constant input and output port names, inlined packet processing code, only AHIR I/O function calls.

USE CASE: EXCERPTS OF GENERATED VHDL

```
...
entity click_ahir_ll_ahir_glue_min_dp is
port(
  ...
  io_dst_in0_ack : in std_logic;
  io_dst_in0_data : out std_logic_vector(31 downto 0);
  io_dst_in0_req : out std_logic;
  io_min_in0_ack : in std_logic;
  io_min_in0_data : in std_logic_vector(15 downto 0);
  io_min_in0_req : out std_logic;
  ...
); end click_ahir_ll_ahir_glue_min_dp;
...
dpe_16 : APInt_S_2 -- tmp7
-- configuration: APIntXor
  generic map(
    colouring => (0 => 0))
  port map(
    ackC => SigmaOut(20 downto 20),
    ackR => SigmaOut(19 downto 19),
    clk => clk,
    reqC => SigmaIn(19 downto 19),
    reqR => SigmaIn(18 downto 18),
    reset => reset,
    x => wire_11,
    y => wire_13,
    z => wire_12);
...
```

USE CASE: EXCERPTS OF GENERATED VHDL

```
...
entity click_ahir_ll_ahir_glue_min_dp is
port(
  ...
  io_dst_in0_ack : in std_logic;
  io_dst_in0_data : out std_logic_vector(31 downto 0);
  io_dst_in0_req : out std_logic;
  io_min_in0_ack : in std_logic;
  io_min_in0_data : in std_logic_vector(15 downto 0);
  io_min_in0_req : out std_logic;
  ...
); end click_ahir_ll_ahir_glue_min_dp;
...
dpe_16 : APInt_S_2 -- tmp7
-- configuration: APIntXor
  generic map(
    colouring => (0 => 0))
  port map(
    ackC => SigmaOut(20 downto 20),
    ackR => SigmaOut(19 downto 19),
    clk => clk,
    reqC => SigmaIn(19 downto 19),
    reqR => SigmaIn(18 downto 18),
    reset => reset,
    x => wire_11,
    y => wire_13,
    z => wire_12);
...
```

Wires representing the read_uintptr() and write_uintptr() functions; leading to previous and next Click elements

USE CASE: EXCERPTS OF GENERATED VHDL

```
...
entity click_ahir_ll_ahir_glue_min_dp is
port(
  ...
  io_dst_in0_ack : in std_logic;
  io_dst_in0_data : out std_logic_vector(31 downto 0);
  io_dst_in0_req : out std_logic;
  io_min_in0_ack : in std_logic;
  io_min_in0_data : in std_logic_vector(15 downto 0);
  io_min_in0_req : out std_logic;
  ...
); end click_ahir_ll_ahir_glue_min_dp;
...
dpe_16 : APInt_S_2 -- tmp7
-- configuration: APIntXor
  generic map(
    colouring => (0 => 0))
  port map(
    ackC => SigmaOut(20 downto 20),
    ackR => SigmaOut(19 downto 19),
    clk => clk,
    reqC => SigmaIn(19 downto 19),
    reqR => SigmaIn(18 downto 18),
    reset => reset,
    x => wire_11,
    y => wire_13,
    z => wire_12);
...
```

Wires representing the read_uintptr() and write_uintptr() functions; leading to previous and next Click elements

Block describing the actual XOR operation on data (the actual operation is defined in a library and uses standard VHDL `xor` on two integers)

CONCLUSIONS

- High-level goal: Study the applicability of modular compiler optimisations for atypical/generated hardware
- Practical goal: Implement a Click-to-NetFPGA tool chain
- Main contributions:
  - Click2LLVM: Dump a Click process memory as LLVM IR constants
  - AHIR: Convert LLVM IR to VHDL
  - The Click/NetFPGA wrapper for the NetFPGA reference NIC design
- Future work:
  - evaluate the performance of the hardware accelerated Click router
  - evaluate different software- and hardware-level optimisations
  - gain more understanding on the possibilities of current compilation and high-level synthesis techniques