CLICK-TO-NETFPGA TOOLCHAIN

TEEMU RINTA-AHO
PEKKA NIKANDER
SAMEER D. SAHASRABUDDHE
JAMES KEMPF

PRESENTATION OUTLINE

- Motivation
- Background (quickly)
  - High Level Synthesis
  - Click Modular Router
  - LLVM
  - AHIR
- Overall Approach
- Current Status
- Future Work
- Demo

MOTIVATION

- Applicability of HLS to packet networking
- Implement a Click-to-NetFPGA toolchain

- No Verilog
- No hardware details
- Unrestricted C++
- Existing Click elements

HIGH LEVEL SYNTHESIS (HLS)

- A high level program → hardware (RTL)
  - C/C++/Systems C → Verilog/VHDL

- Current tools: limitations and “features”
  - For hardware professionals; for “writing hardware” in C
  - Subset of C/C++, excluding e.g.:
    - dynamic memory
    - function pointers and virtual functions
    - recursion
  - Cannot handle existing software-oriented code
    - e.g. Click elements
  - Closed source commercial products

CLICK MODULAR ROUTER

- Software routers from elements
  - Open source, by Eddie Kohler (MIT)
- Linux/BSD/Darwin userspace & Linux kernel
  - Also OpenWRT etc.
  - (BTW, we are working on a FreeBSD kernel version)
- 100+ existing elements
- Easy to add new elements
- Easy to build a custom routers

LOW LEVEL VIRTUAL MACHINE (LLVM)

- Compiler toolkit; GCC replacement
  - Open source, by Chris Lattner (UIUC → Apple)
  - Outperforms GCC in many (but not all) ways

- Frontends, tools, optimisers, backends

- Easy to write new compiler passes
- Easy to write backends (and, maybe, front ends)

- Represents code as SSA (Single Static Assignment)
  - An abstract, assembler-like form, with unlimited registers
- Can perform global optimisations (after linking)

AHIR
"A HARDWARE INTERMEDIATE REPRESENTATION"

- LLVM backend for generating VHDL
  - To-be open source, by Sameer Saharsabuddhe (IIT Bombay)
  - Factorises the system into control, data, and storage
    - Supports scalable optimisations and analyses

- An LLVM backend
  - Current limitations: no recursion or function pointers
  - An LLVM IR function corresponds to a VHDL module

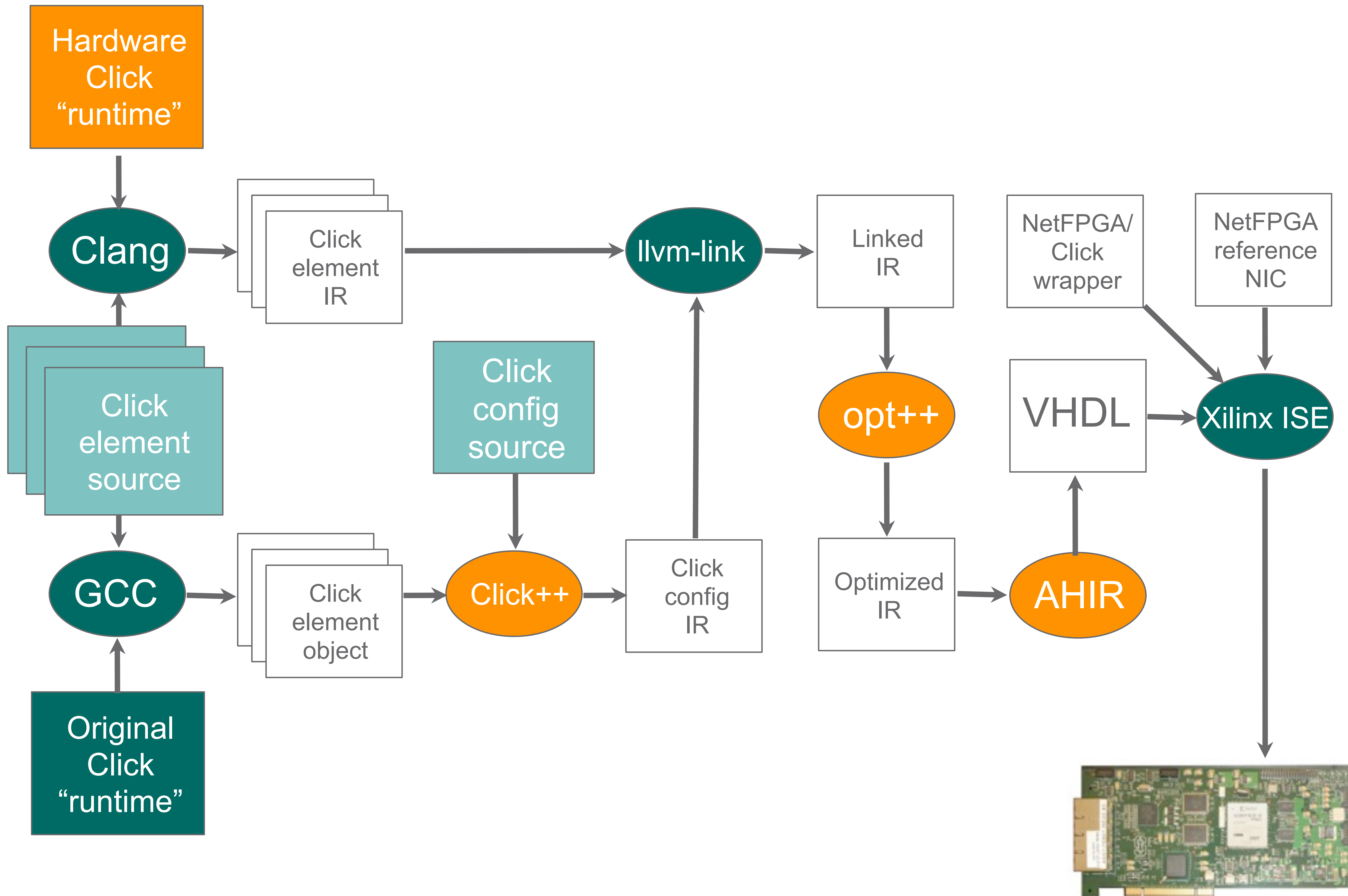
- Design = Set of modules with I/O channels
  - I/O through a simple IPC library, resembling Unix pipes

PRESENTATION OUTLINE

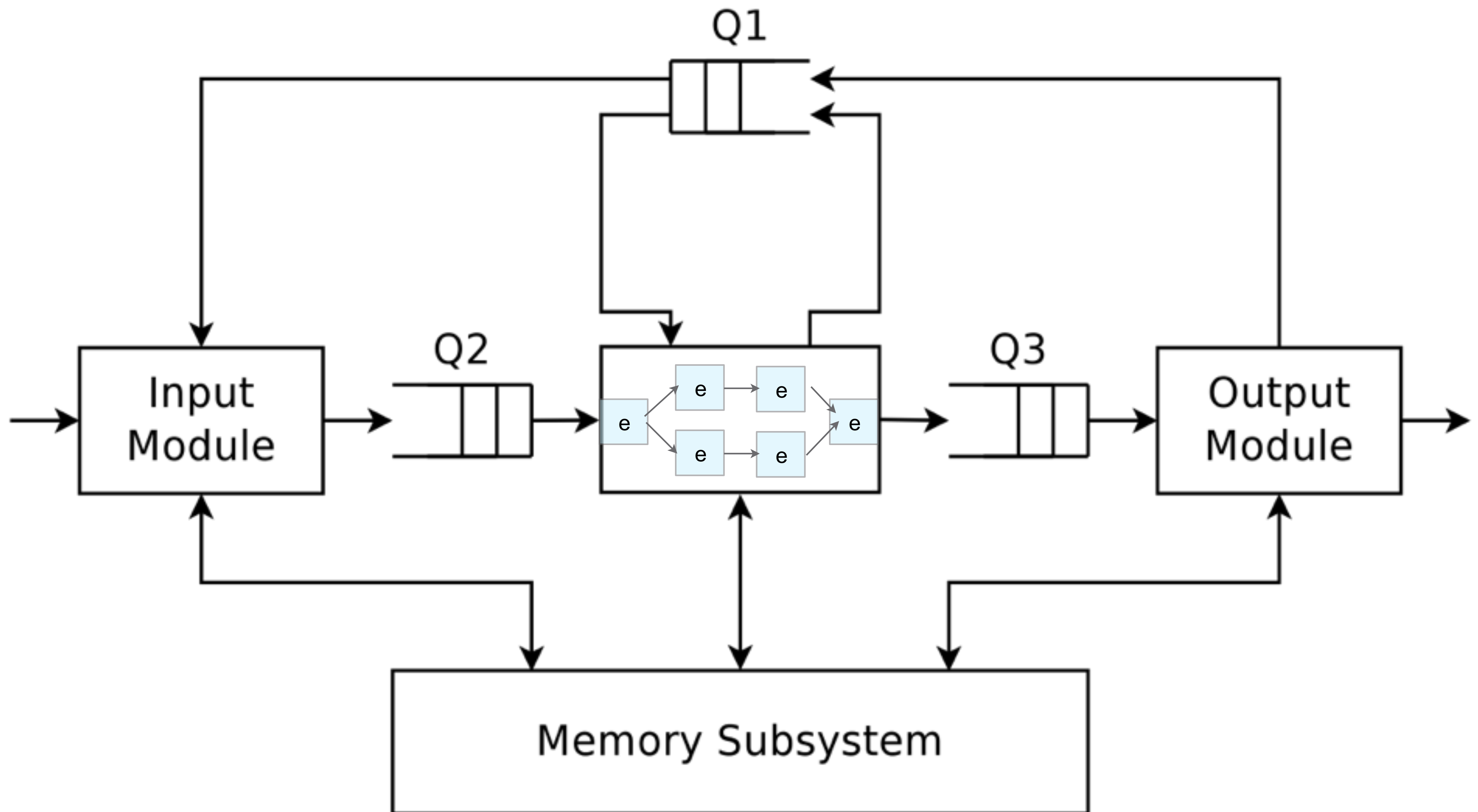
- Motivation
- Background (quickly)
  - High Level Synthesis
  - Click Modular Router
  - LLVM
  - AHIR
- Overall Approach
- Current Status
- Future Work
- Demo

OVERALL APPROACH

- Click C++ → "hardware friendly" LLVM IR → VHDL
  - A "hardware runtime" library and header files
    - E.g. replace heap allocation with table-based allocation
  - LLVM-based code optimisations
    - Remove virtual function calls
    - Aggressive constant propagation
- Click configuration → LLVM IR (via C++)
  - Use Click to set up a "Click router" in compiler memory
  - Dump the memory as constants, expressed in LLVM IR


Hardware Click "runtime"
Clang
Click element IR
llvm-link
Linked IR
NetFPGA/ Click wrapper
NetFPGA reference NIC
Click element source
Click config source
opt++
VHDL
Xilinx ISE
GCC
Click element object
Click++
Click config IR
Optimized IR
AHIR
Original Click "runtime"

VHDL NETFPGA/CLICK WRAPPER



CURRENT STATUS

- A very early prototype
  - Still contains lots of bailing wire and chewing gum
  - About 9000 lines of code:
    - 2 KLoC new code
    - 7 KLoC of modified Click runtime headers and libraries
- Works for simple Click configurations
  - Only sequential element chains
    - each element has one input and one output port
    - branching chains soon
  - Limited configuration option processing
    - 4-5 data types out of the ~40 types supported
  - No runtime re-configuration of router or elements
  - No cloning of packets

FUTURE WORK

- September 2010:
  - Complete the VHDL NetFPGA/Click wrapper
  - Add support for branching Click element chains
  - Add support for more configuration option data types
  - Synthetise some useful examples
    - A minimal IP/UDP host with ARP, ICMP and UDP echo
    - A minimal Ethernet bridge
- Until early 2011:
  - Add support for more configuration option data types
  - Refactor the Click-based frontend not to use C++
    - Generate LLVM IR directly, via augmenting clang AST
  - Deal with all the unknown problems that will surface
  - Syntetise non-trivial examples

DEMO TIME

EXAMPLE 1: CLICK CONFIGURATION

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

src    :: FromFPGA;
simple :: SimpleIdle;

src -> simple;
```

Include the FromFPGA element

A standard Click element which simply discards all packets

EXAMPLE 1: CONFIGURATION.CC

```
/*** This file was generated with 'click -c' ***/
#include "click/config.h"
#include "fromfpga.hh"
#include "simple/simpleidle.hh"

extern "C" {
  int __cxa_atexit(void (*)(void*), void*, void*);
  void ahir_glue_src(void);
  void ahir_glue_simple(void);
}
int __cxa_atexit(void (*)(void*), void*, void*) { return 0; }
extern Packet *const packet_out;

/* src */
Element::Port const        src_in_0   = Element::Port(0, "netfpga_in");
Element::Port const *const src_ins[]  = { &src_in_0 };
Element::Port const        src_out_0  = Element::Port(0, "src_out0");
Element::Port const *const src_outs[] = { &src_out_0 };
Element::Port const *const *const src_ports_io[]   = { src_ins, src_outs };
Element::Port const *const *const *const src_ports = src_ports_io;
FromFPGA const src = FromFPGA(src_ports);

/* simple */
Element::Port const        simple_in_0   = Element::Port(0, "src_out0");
Element::Port const *const simple_ins[]  = { &simple_in_0 };
Element::Port const        simple_out_0  = Element::Port(0, "simple_out0");
Element::Port const *const simple_outs[] = { &simple_out_0 };
Element::Port const *const *const simple_ports_io[]   = { simple_ins, simple_outs };
Element::Port const *const *const *const simple_ports = simple_ports_io;
SimpleIdle const simple = SimpleIdle(simple_ports);

void ahir_glue(const Element &e);

void ahir_glue_src(void) { ahir_glue(src); }
void ahir_glue_simple(void) { ahir_glue(simple); }
```

Hacks for naming & handling dynamic C++ constructors

Static connections between Click elements

Top-level functions for AHIR

EXAMPLE 1: GLUE.CC

```
#include "fromfpga.hh"
#include "simple/simpleidle.hh"

void ahir_glue(const Element &e) {
    uint64_t *p;

    uint32_t p_int = read_uint32(e._ports[0][0][0]._ahir_port_name);
    p = (uint64_t *)(p_int);

    Packet packet_in(p);    // Construct a “phantom” packet

    e.push(0, &packet_in);  // Invoke the element on the packet
}
```

- The AHIR top level function for each Click element
- Currently only **`push`** and single input/output ports
- Adding **`pull`** and multiple ports is being implemented

EXAMPLE 1: AHIR_GLUE_SRC()

Version A: Marking packet liveliness in the first NetFPGA 64-bit word

```
@"\01LC" = internal constant [11 x i8] c"netfpga_in\00"
@"\01LC1" = internal constant [9 x i8] c"src_out0\00"

define void @ahir_glue_src() {
  %0 = tail call i32 @read_uint32(i8* getelementptr
                           inbounds ([11 x i8]* @"\01LC", i64 0, i64 0))
  %1 = inttoptr i32 %0 to i64*
  %2 = load i64* %1, align 1
  %3 = and i64 %2, -36028797018964000  # 0xFF7FFFFFFFFFFFFF
  store i64 %3, i64* %1, align 1
  tail call void @write_uint32(i8* getelementptr
                           inbounds ([9 x i8]* @"\01LC1", i64 0, i64 0), i32 %0)
  ret void
}
```

Version B: Marking liveliness at Packet::_flags (which get optimized away)

```
define void @ahir_glue_src() {
  %0 = tail call i32 @read_uint32(i8* getelementptr
                           inbounds ([11 x i8]* @"\01LC", i64 0, i64 0))
  tail call void @write_uint32(i8* getelementptr
                           inbounds ([9 x i8]* @"\01LC1", i64 0, i64 0), i32 %0)
  ret void
}
```

EXAMPLE 1: AHIR_GLUE_SIMPLE()

Version A: Marking packet liveliness in the first NetFPGA 64-bit word

```
@"\01LC1" = internal constant [9 x i8] c"src_out0\00"
@"\01LC2" = internal constant [11 x i8] c"free_queue\00"

define void @ahir_glue_simple() {
  %0 = tail call i32 @read_uint32(i8* getelementptr
               inbounds ([9 x i8]* @"\01LC1", i64 0, i64 0))
  %1 = inttoptr i32 %0 to i64*
  %2 = load i64* %1, align 1
  %3 = or i64 %2, 36028797018963968 # 0x0080000000000000
  store i64 %3, i64* %1, align 1
  tail call void @write_uint32(i8* getelementptr
               inbounds ([11 x i8]* @"\01LC2", i64 0, i64 0), i32 %0)
  ret void
}
```

Packet gets marked as killed

EXAMPLE 1: AHIR VHDL DATAPATH

```
entity ahir_glue_src_dp is
  port(
    SigmaIn : in BooleanArray(13 downto 1);
    SigmaOut : out BooleanArray(13 downto 1);
    call_ack : out std_logic;
    call_data : in std_logic_vector(0 downto 0);
    call_req : in std_logic;
    call_tag : in std_logic_vector;
    clk : in std_logic;
    io_netfpga_in_ack : in std_logic;
    io_netfpga_in_data : in std_logic_vector(31 downto 0);
    io_netfpga_in_req : out std_logic;
    io_src_out0_ack : in std_logic;
    io_src_out0_data : out std_logic_vector(31 downto 0);
    io_src_out0_req : out std_logic;
    lc_ack : in std_logic_vector(7 downto 0);
    lc_data : in std_logic_vector(63 downto 0);
    lc_req : out std_logic_vector(7 downto 0);
    lc_tag : in std_logic_vector(63 downto 0);
    lr_ack : in std_logic_vector(7 downto 0);
    lr_addr : out std_logic_vector(127 downto 0);
    lr_req : out std_logic_vector(7 downto 0);
    lr_tag : out std_logic_vector(63 downto 0);
    reset : in std_logic;
    return_ack : in std_logic;
    return_data : out std_logic_vector(0 downto 0);
    return_req : out std_logic;
    return_tag : out std_logic_vector;
    sr_ack : in std_logic_vector(7 downto 0);
    sr_addr : out std_logic_vector(127 downto 0);
    sr_data : out std_logic_vector(63 downto 0);
    sr_req : out std_logic_vector(7 downto 0);
    sr_tag : out std_logic_vector(63 downto 0));
end ahir_glue_src_dp;
```

```
entity ahir_glue_simple_dp is
  port(
    SigmaIn : in BooleanArray(18 downto 1);
    SigmaOut : out BooleanArray(18 downto 1);
    call_ack : out std_logic;
    call_data : in std_logic_vector(0 downto 0);
    call_req : in std_logic;
    call_tag : in std_logic_vector;
    clk : in std_logic;
    io_free_queue_put_ack : in std_logic;
    io_free_queue_put_data : out std_logic_vector(31 downto 0);
    io_free_queue_put_req : out std_logic;
    io_src_out0_ack : in std_logic;
    io_src_out0_data : in std_logic_vector(31 downto 0);
    io_src_out0_req : out std_logic;
    lc_ack : in std_logic_vector(15 downto 0);
    lc_data : in std_logic_vector(127 downto 0);
    lc_req : out std_logic_vector(15 downto 0);
    lc_tag : in std_logic_vector(127 downto 0);
    lr_ack : in std_logic_vector(15 downto 0);
    lr_addr : out std_logic_vector(255 downto 0);
    lr_req : out std_logic_vector(15 downto 0);
    lr_tag : out std_logic_vector(127 downto 0);
    reset : in std_logic;
    return_ack : in std_logic;
    return_data : out std_logic_vector(0 downto 0);
    return_req : out std_logic;
    return_tag : out std_logic_vector;
    sr_ack : in std_logic_vector(15 downto 0);
    sr_addr : out std_logic_vector(255 downto 0);
    sr_data : out std_logic_vector(127 downto 0);
    sr_req : out std_logic_vector(15 downto 0);
    sr_tag : out std_logic_vector(127 downto 0));
end ahir_glue_simple_dp;
```

EXAMPLE 2: CLICK CONFIGURATION

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

src     :: FromFPGA;
dst     :: ToFPGA;
minimal :: Minimal;

src -> minimal -> dst;
```

Include the FromFPGA and ToFPGA elements

A Click element which modifies packet data by **adding 1 to the first word**

EXAMPLE 2: AHIR_GLUE_MINIMAL()

```
@"\01LC1" = internal constant [9 x i8] c"src_out0\00"
@"\01LC2" = internal constant [13 x i8] c"minimal_out0\00"

define void @ahir_glue_minimal() {
  %0 = tail call i32 @read_uint32(i8* getelementptr
             inbounds ([9 x i8]* @"\01LC1", i64 0, i64 0))
  %1 = inttoptr i32 %0 to i64*
  %2 = getelementptr i64* %1, i64 1
  %3 = load i64* %2, align 1
  %4 = add i64 %3, 1
  store i64 %4, i64* %2, align 1
  tail call void @write_uint32(i8* getelementptr
             inbounds ([13 x i8]* @"\01LC2", i64 0, i64 0), i32 %0)
  ret void
}
```

Minimal element adds +1 to the first word of the packet’s data and passes the packet forward to port “minimal_out0”