GENERATING HARDWARE: CLICK-TO-NETFPGA TOOLCHAIN USING LLVM

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

(PRESENTED BY ERIK RUBOW)

PRESENTATION OUTLINE

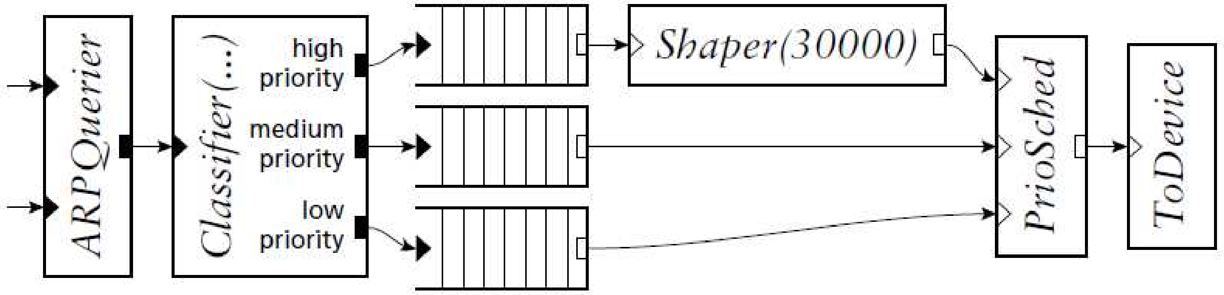
- Motivation
- Background
  - Click Modular Router
  - Stanford NetFPGA prototyping board
  - High Level Synthesis (HLS)
  - AHIR
- Overall Approach
- Current Status
- Future Work
- Summary

MOTIVATION

- Study the applicability of HLS to packet networking
- Implement a Click-to-NetFPGA toolchain
- More generally: 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?

CLICK MODULAR ROUTER

- Software platform for building various kinds of packet-processing systems, or “routers”
  - – Open source, by Eddie Kohler (MIT/UCLA)
- Runs in Linux/BSD/Darwin userspace & Linux kernel
  - – (BTW, we are also working on a FreeBSD kernel version)
- Modular framework for composing “elements” into routers
- 100+ existing elements, e.g. ARPResponder, Classifier
- Easy to add new elements
- Configuration language for defining a system by paramaterizing and interconnecting elements



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 coming early next year
  - –4 x 10G Ethernet interface, bigger FPGA, faster PCIe interface

HIGH LEVEL SYNTHESIS (HLS)

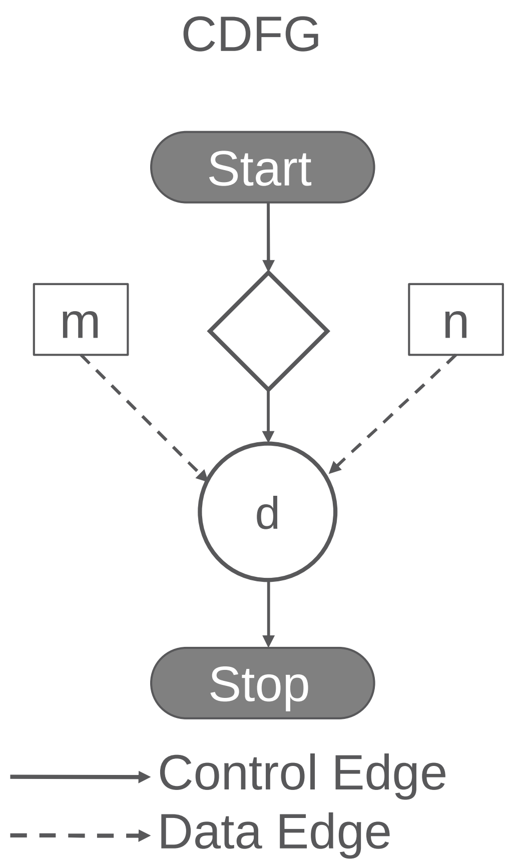
- A high level program → hardware (RTL)
  - Typically: C/C++/SystemC → Verilog/VHDL
- Limitations of currently available tools:
  - Designed for hardware professionals; for “writing hardware” in C
  - Support only a subset of C/C++, excluding e.g.:
    - dynamic memory allocation
    - function pointers and virtual functions
    - recursion
  - Cannot handle existing software-oriented code
    - e.g. Click elements
  - Closed-source commercial products

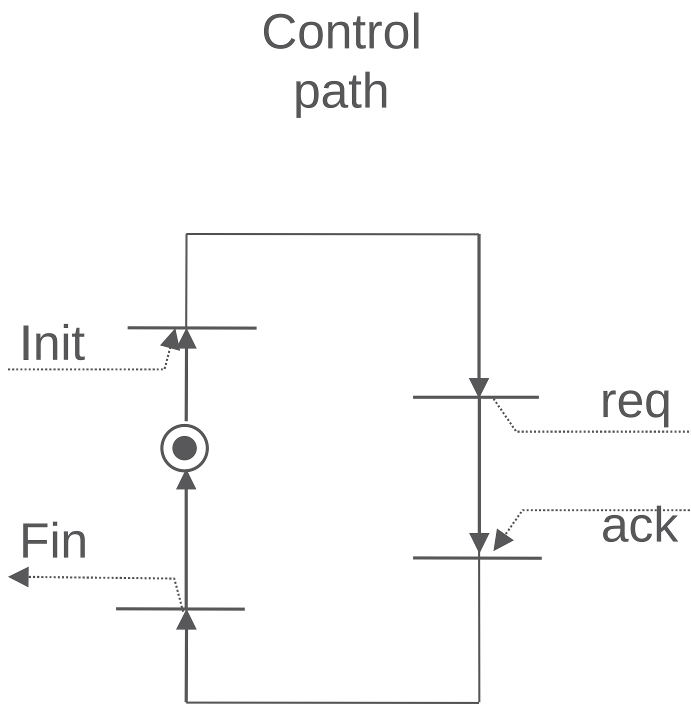
AHIR

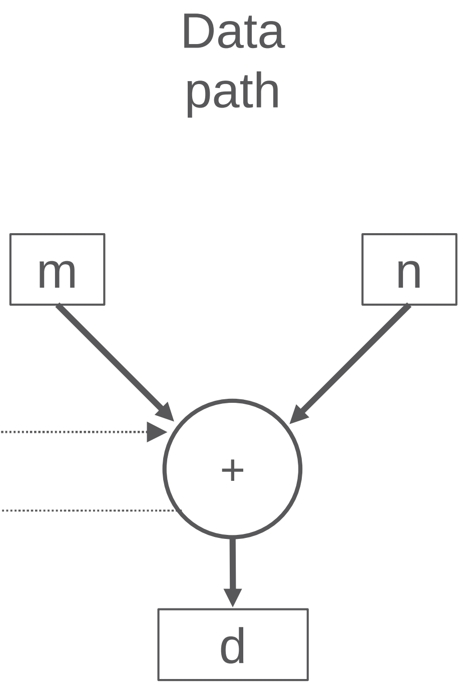
- “A Hardware Intermediate Representation”
  - To-be open source, by Sameer Sahasrabuddhe (IIT Bombay)
  - A convenient midway point between software and hardware
  - Factorises the system into control, data, and storage
    - Control is modeled as a petri net
    - Supports scalable optimisations and analyses
  - Allows mapping LLVM IR -> AHIR -> VHDL
  - Generates a VHDL module out of each LLVM IR function
  - Current limitations: no recursion or function pointers, otherwise full C
  - Has a built-in library for asynchronous I/O between modules

AN AHIR EXAMPLE 1
CONVERTING AN LLVM INSTRUCTION TO VHDL

- Simple Addition Example
- C code: d = m + n







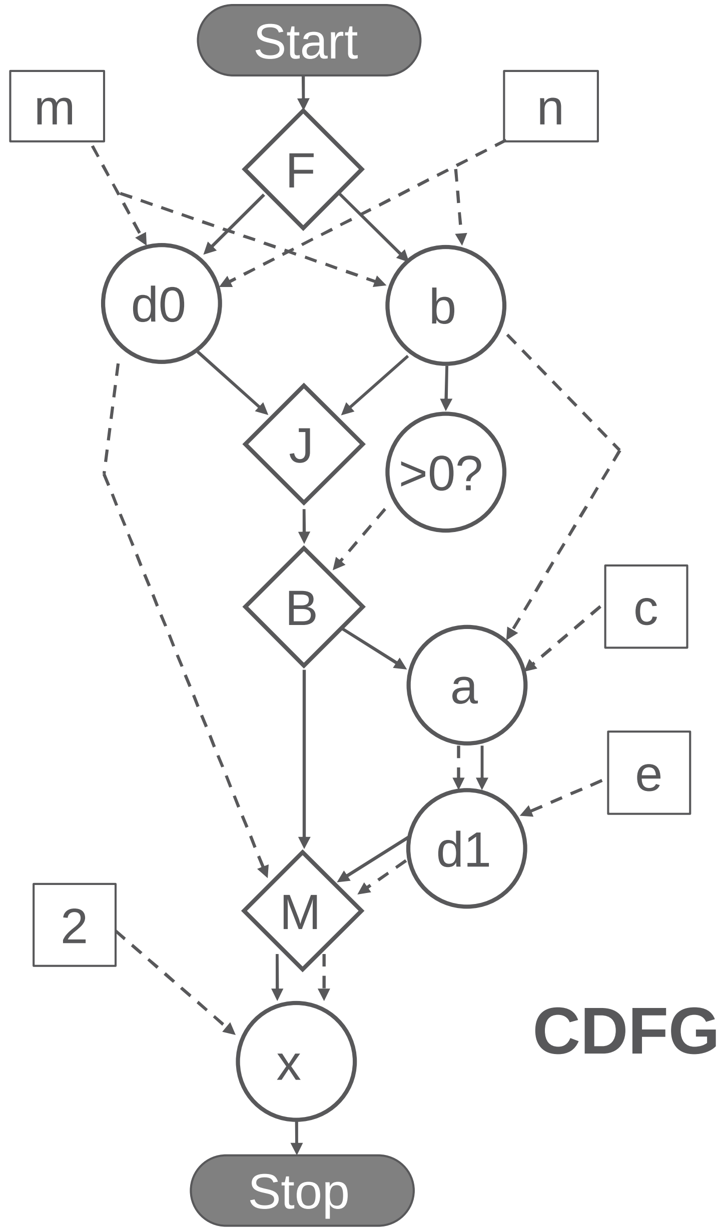
AN AHIR EXAMPLE 2
CONVERTING LLVM INSTRUCTIONS TO VHDL

C code:

```
d = m + n;
b = m - n;
if (b > 0) {
  a = b + c;
  d = e + a;
}
x = d + 2;
```

SSA version:

```
d0 = m + n;
b = m - n;
if (b > 0) {
  a = b + c;
  d1 = e + a;
}
d2 = phi(d0, d1);
x = d2 + 2;
```



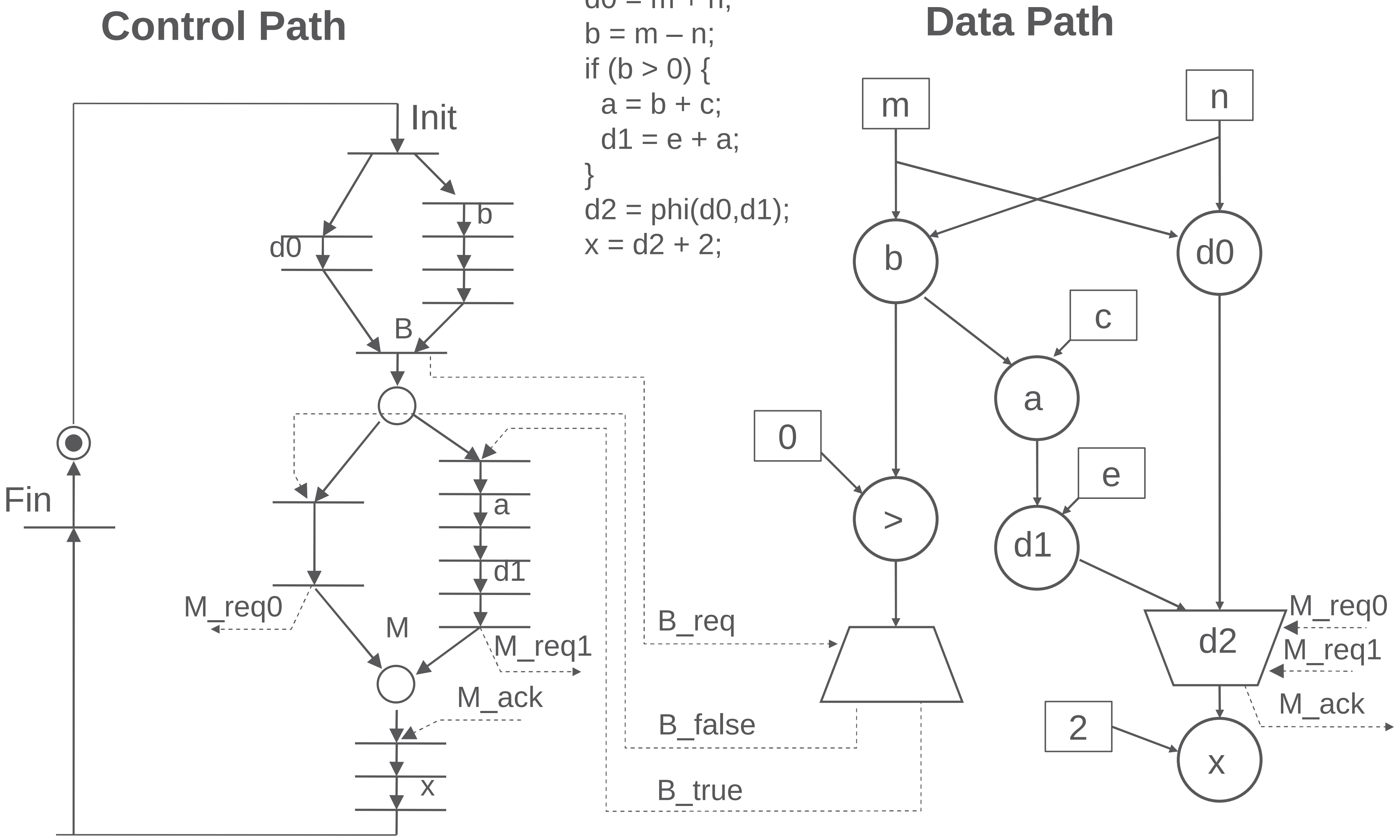
CDFG

Control Edge
Data Edge

AN AHIR EXAMPLE 2

CONVERTING LLVM INSTRUCTIONS TO VHDL

```
d0 = m + n;
b = m – n;
if (b > 0) {
  a = b + c;
  d1 = e + a;
}
d2 = phi(d0,d1);
x = d2 + 2;
```



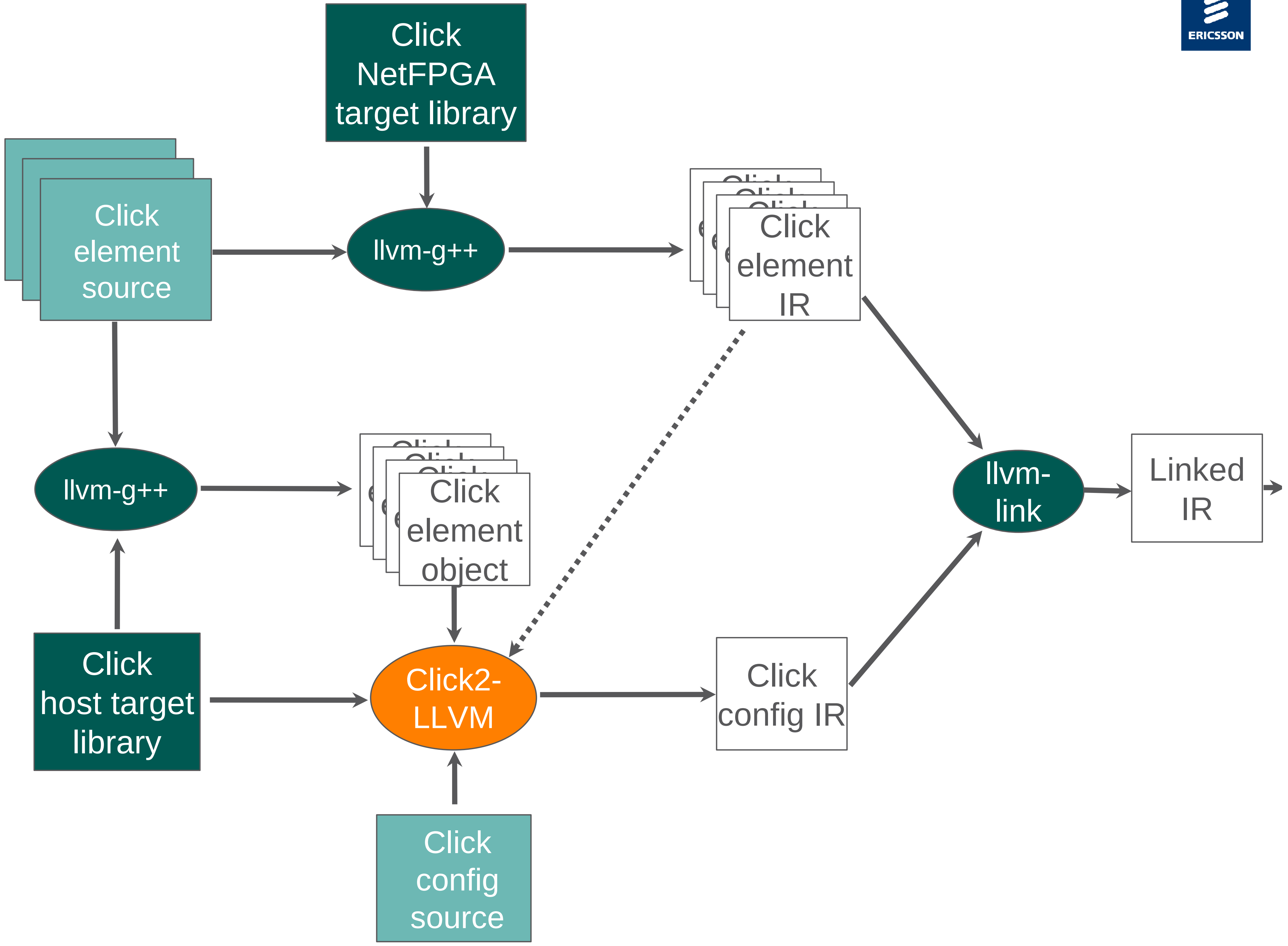
PRESENTATION OUTLINE

OVERALL APPROACH

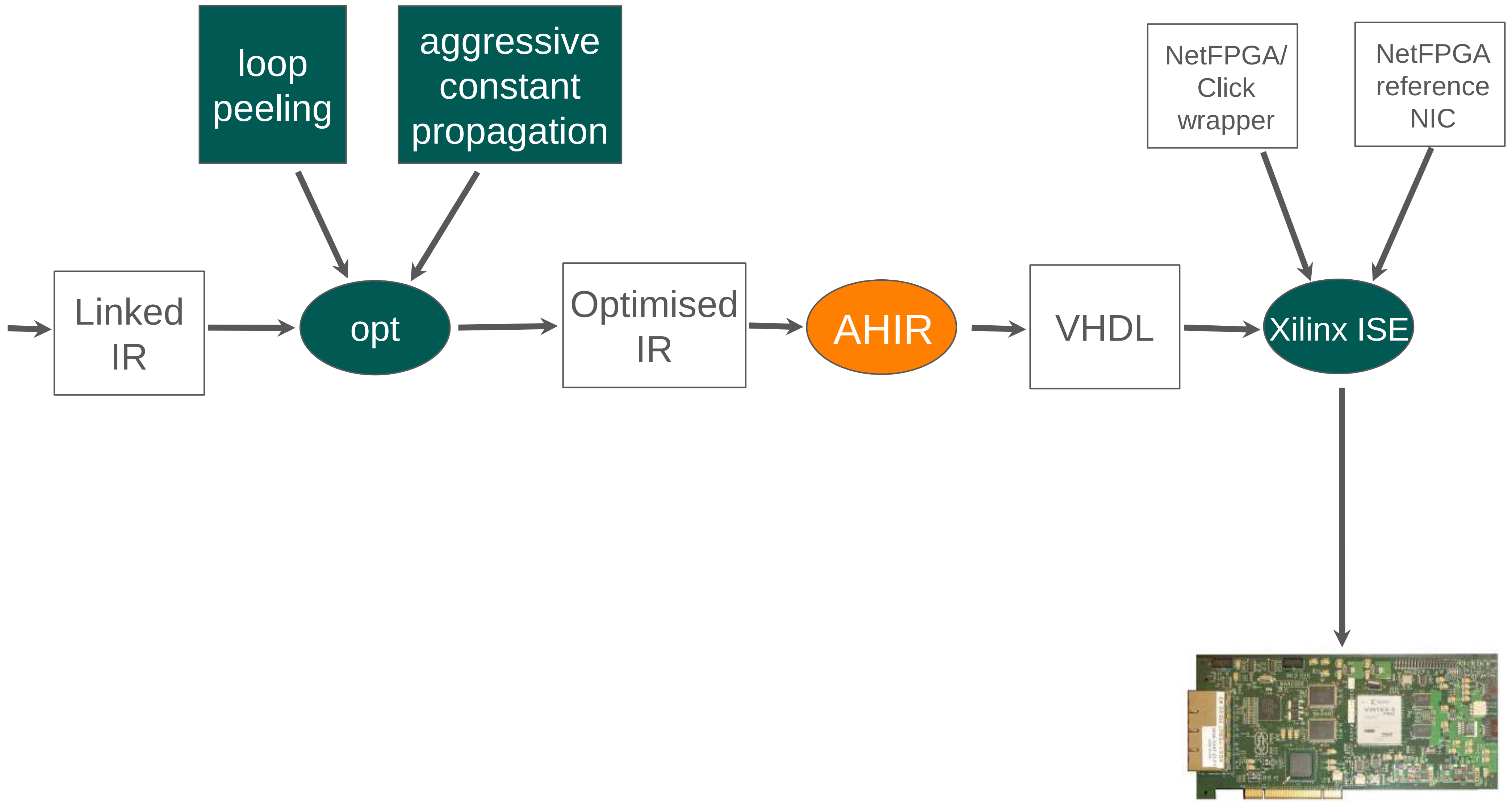
- In software Click, the compiled code includes the implementations of all elements in the library
  - Includes lots of initialization and configuration parsing
  - Too much code to realistically turn into hardware
- For hardware implementation, we want a chunk of hardware for each element instance in a particular design
  - Only include what we need for processing packets; cut out the “fat”
- This is possible: there is an initialization phase followed by a packet processing phase
- Need to transform the code into something a HLS tool can handle
  - E.g. replacing virtual function calls with static function calls
- Need to integrate the generated hardware with a known hardware environment

OVERALL APPROACH

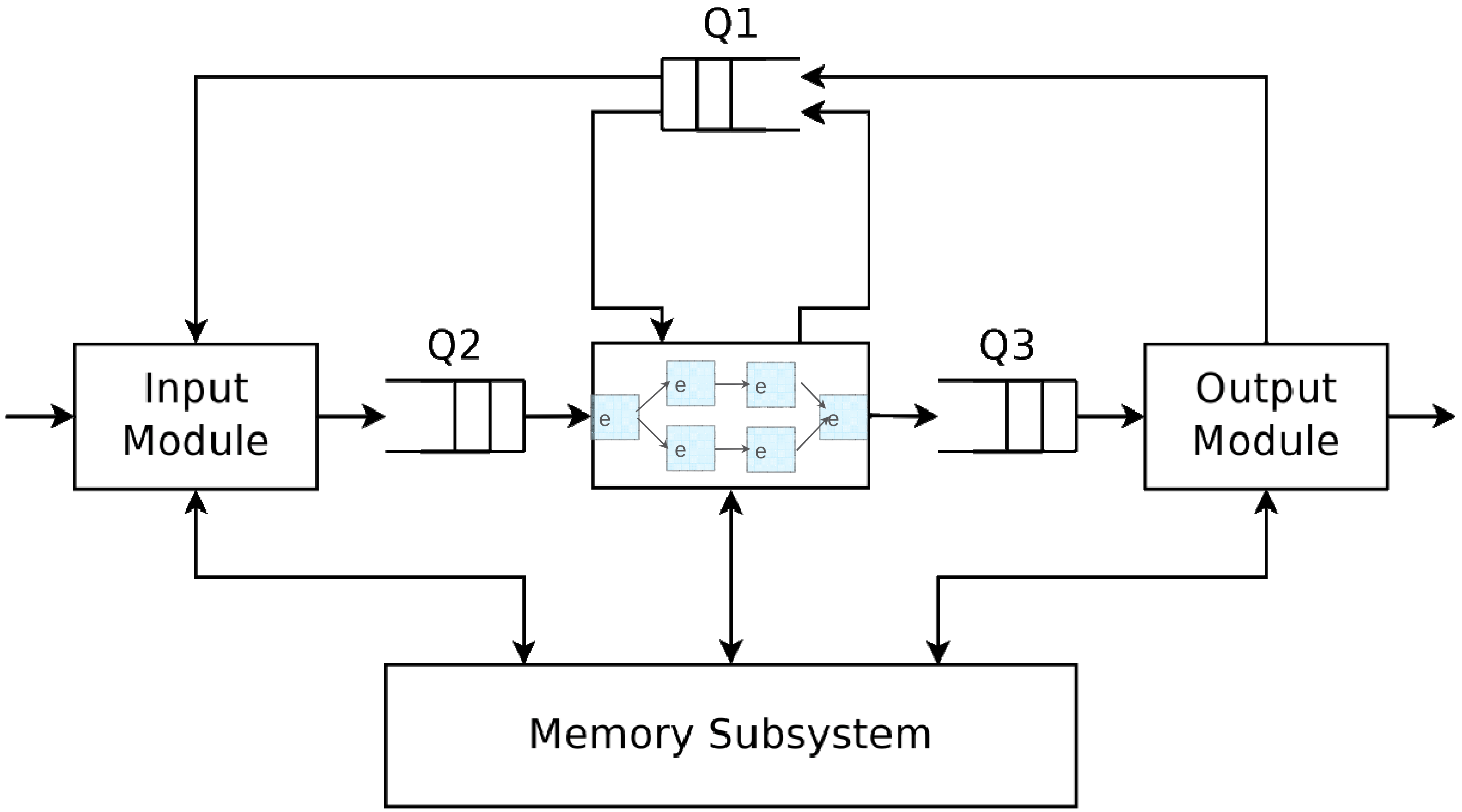
- Click C++ → “hardware friendly” LLVM IR → VHDL
  - Modified Click library for new “target”, similar to BSD/Linux specific differences
    - E.g. use table-based memory allocation instead of skb/heap-based
  - Click2LLVM: Click configuration → LLVM IR
    - Use Click library functions to set up a “Click router” in memory
    - Dump the memory as a set of constants, expressed in LLVM IR
  - LLVM IR → “hardware friendly” LLVM IR
    - Loop peeling
    - Aggressive constant propagation and dead code elimination
    - Together result 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


loop peeling
aggressive constant propagation
NetFPGA/ Click wrapper
NetFPGA reference NIC
Linked IR
opt
Optimised IR
AHIR
VHDL
Xilinx ISE

VHDL NETFPGA/CLICK WRAPPER



NETFPGA WRAPPER FOR THE CLICK/AHIR SYSTEM

› NetFPGA process 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 contains elements from Click configuration
  - Click elements may create or destroy packets using direct access to the free queue

EXAMPLE: TEST.CLICK

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

src :: FromFPGA;
ehm :: EtherMirror;
dst :: ToFPGA;

src -> ehm -> dst;
```

test.click

- The Click configuration consists of three elements:
  - one standard element
    - EtherMirror
      - Swaps ethernet addresses
  - two elements specific to our Click-NetFPGA wrapper
    - FromFPGA and ToFPGA
      - Map between wrapper-specific and Click-specific metadata

EXAMPLE: ORIGINAL CLICK C++

```
Packet * EtherMirror::simple_action(Packet *p){
    if (WritablePacket *q = p->uniqueify()) {
        click_ether *ethh = reinterpret_cast<click_ether *>(q->data());
        uint8_t tmpa[6];
        memcpy(tmpa, ethh->ether_dhost, 6);
        memcpy(ethh->ether_dhost, ethh->ether_shost, 6);
        memcpy(ethh->ether_shost, tmpa, 6);
        return q;
    }
    else
        return 0;
}
```

excerpt from ethermirror.cc

› simple_action() is the only method defined in EtherMirror element, everything else is inherited from Element

› With standard Click, when a downstream Click element calls Element::port::push(p) on a port connected to EtherMirror, simple_action() is called, then push() on the next element is called, and so on..

EXAMPLE: CLICK2LLVM

- The toolchain (Click2LLVM + llvm-link + opt) produces a self-contained LLVM module out of a Click configuration:
  - – Output from Click2LLVM + llvm-link:
    - click-ahir.ll (16,761 lines)
  - – Running that through opt:
    - click-ahir-opt.ll (171 lines)
- The LLVM module contains a function per Click element
  - –named “ahir_glue_<element_name>”
- Click library calls inlined due to optimizations
- Modified Click library introduces hooks for AHIR/NetFPGA
  - –E.g. Packet::kill(p)   => write_uintptr(“free_queue_put”, p)
  - –write_uintptr() is later translated by AHIR to a VHDL write to the “free queue” hardware FIFO

EXAMPLE: OPTIMIZED LLVM IR

```
define void @ahir_glue_ehm() ssp {
  %0 = tail call i64 @read_uintptr(i8* getelementptr inbounds ([8 x i8]* @3, i64 0, i64 0))
  %1 = inttoptr i64 %0 to %struct.Packet*
  %2 = getelementptr inbounds %struct.Packet* %1, i64 0, i32 3
  %3 = load i8** %2, align 8
  %tmp8.i = load i8* %3, align 1
  %.19.i = getelementptr inbounds i8* %3, i64 1
  %tmp10.i = load i8* %.19.i, align 1
  %.211.i = getelementptr inbounds i8* %3, i64 2
  %tmp12.i = load i8* %.211.i, align 1
  %.313.i = getelementptr inbounds i8* %3, i64 3
  %tmp14.i = load i8* %.313.i, align 1
  %.415.i = getelementptr inbounds i8* %3, i64 4
  %tmp16.i = load i8* %.415.i, align 1
  %.517.i = getelementptr inbounds i8* %3, i64 5
  %tmp18.i = load i8* %.517.i, align 1
  %4 = getelementptr inbounds i8* %3, i64 6
  tail call void @llvm.memcpy.p0i8.p0i8.i64(i8* %3, i8* %4, i64 6, i32 1, i1 false) nounwind
  store i8 %tmp8.i, i8* %4, align 1
  %.1.i = getelementptr inbounds i8* %3, i64 7
  store i8 %tmp10.i, i8* %.1.i, align 1
  %.2.i = getelementptr inbounds i8* %3, i64 8
  store i8 %tmp12.i, i8* %.2.i, align 1
  %.3.i = getelementptr inbounds i8* %3, i64 9
  store i8 %tmp14.i, i8* %.3.i, align 1
  %.4.i = getelementptr inbounds i8* %3, i64 10
  store i8 %tmp16.i, i8* %.4.i, align 1
  %.5.i = getelementptr inbounds i8* %3, i64 11
  store i8 %tmp18.i, i8* %.5.i, align 1
  tail call void @write_uintptr(i8* getelementptr inbounds ([1 x %"struct.Element::Port"]* @1,
                               i64 0, i64 0, i32 0, i64 0), i64 %0)
  ret void
}
```

excerpt from click-ahir-opt.ll showing ahir_glue_ehm()

EXAMPLE: RESULTING VHDL

- ahir_glue_ehm_cp.vhdl (control path, 254 lines)
- ahir_glue_ehm_dp.vhdl (data path, 130 lines)
- ahir_glue_ehm_ln.vhdl (link layer, 29 lines)

```
entity ahir_glue_ehm_dp is
  port(
    SigmaIn : in BooleanArray(4 downto 1);
    SigmaOut : out BooleanArray(4 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_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_src_out0_ack : in std_logic;
    io_src_out0_data : in std_logic_vector(31 downto 0);
    io_src_out0_req : out std_logic;
    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);
end ahir_glue_ehm_dp;
```

excerpt from ahir_glue_ehm_dp.vhdl

- Port src_out0 leads to FromFPGA
- Port dst_in0 leads to ToFPGA

- C++ “push/pull” calls are now replaced by hardware FIFOs and all Click elements are running in parallel, processing different packets

CURRENT STATUS

- An early prototype; work in progress
  - About 2200 lines of code:
    - 2000 lines of new code
    - 200 lines of target-specific Click modifications
- Works for simple Click configurations
  - No runtime re-configuration of router or elements
  - Limited configuration option processing
    - ~10 out of the ~40 Click-specific data types supported
  - No cloning of packets yet
  - No performance evaluations yet
- Soon available as an “alpha” release to interested parties

FUTURE WORK

- ›Late 2010:
  - –Complete the basic tool chain features
    - ›Loop-peeling, LLVM metadata-based type annotation, …
  - –Add support for more Click-specific data types
  - –Synthetise some “real life” examples
    - ›A minimal IP/UDP host with ARP, ICMP and UDP echo
    - ›A minimal Ethernet bridge
- ›Early 2011:
  - –Deal with all the unknown problems that will surface
  - –Synthetise non-trivial examples

SUMMARY

›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

›Current status: early prototype, work in progress

– Able to generate hardware from trivial Click configurations