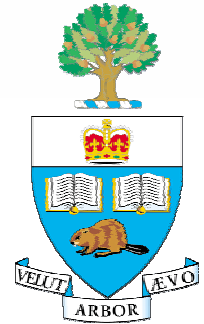




ROGERS
DEPARTMENT OF
ELECTRICAL
& COMPUTER
ENGINEERING
UNIVERSITY of TORONTO



A Verilog RTL Synthesis Tool for Heterogeneous FPGAs

Peter Jamieson and Jonathan Rose

The Edward S. Rogers Sr. Department of Electrical
and Computer Engineering

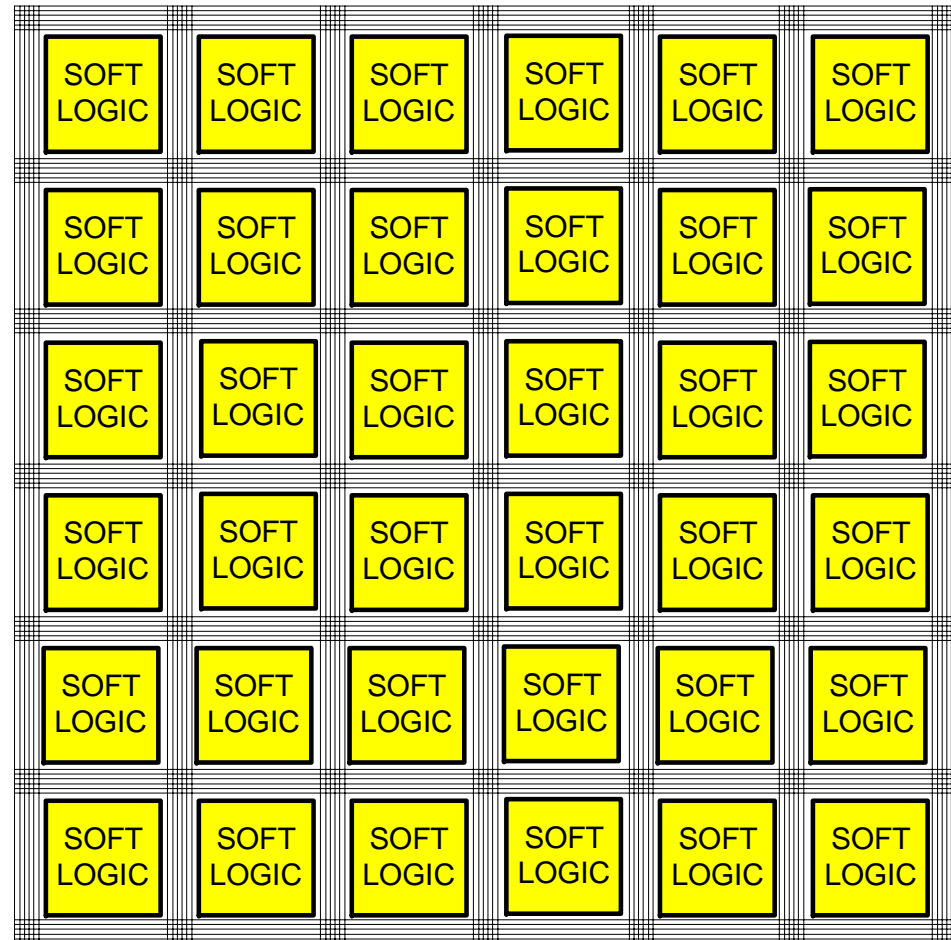
University of Toronto



Homogeneous FPGAs

Consist of:

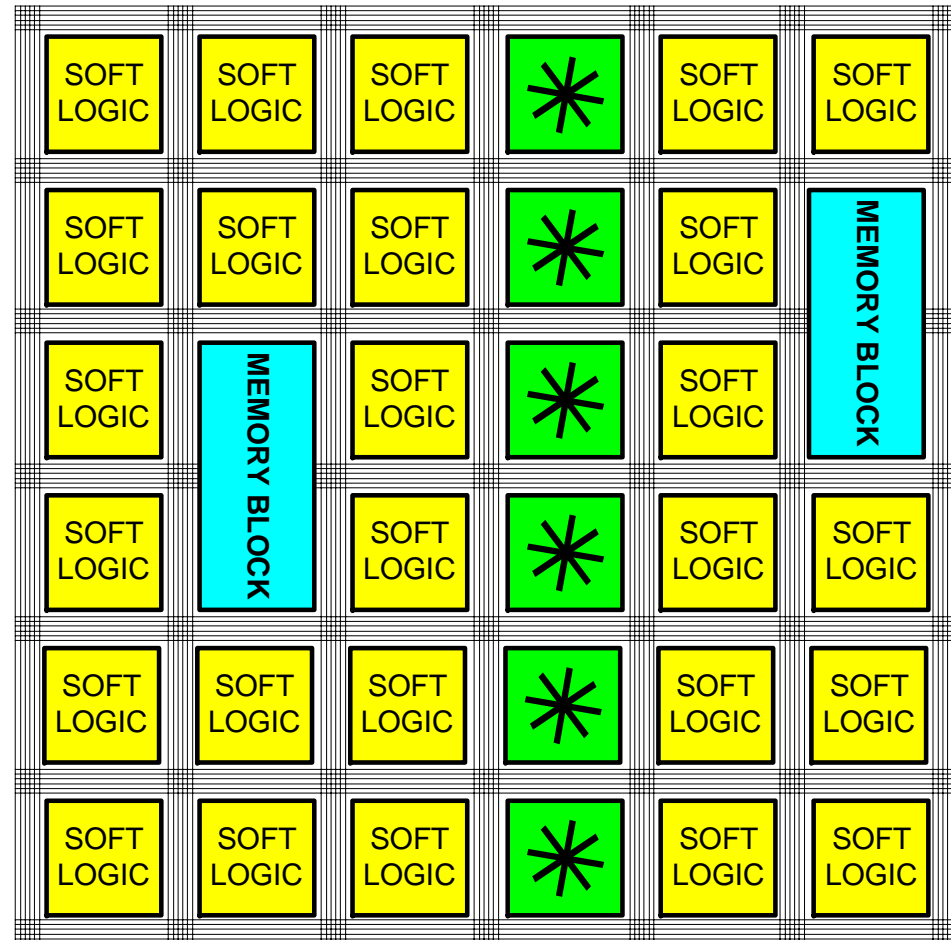
1. Programmable logic and routing
 - soft logic fabric
 - Basic logic unit
 - Programmable Routing



Heterogeneous FPGAs

Consist of:

1. Programmable logic and routing
 - soft logic fabric
 - **hard structures**
2. Dedicated **hard structures**
 - e.g. multiplier
 - e.g. memory block





Fundamental Trade-off

- Hard structures provide benefit *when used*
 - Faster
 - Smaller
 - Consume less power





However...

- If hard structure *not* used
wasted Silicon
 - Routing resources wasted
 - 70-90% of FPGA area occupied by routing





Motivation of This Work

- Long term: derive more benefit from hard structures
 - By exploring the fundamental trade-off
- Academic CAD flows currently can't target these structures





State-of-the-Art CAD Flows

- Existing Front-end Synthesis tools do target heterogeneous FPGAs
 - Altera's Quartus
 - Xilinx 'ISE
 - Mentor's LeonardoSpectrum
 - Synplicity's Synplify
 - Synopsys' Design Compiler FPGA
 - Magma's Blast FPGA





Goals of this Work

1. Front-end tool to map to hard structures
2. Achieve comparable results to Industrial Front-end Synthesis
3. Deliver open source for academic community





Our Tool

Called “Odin”

- maps Verilog HDL designs to heterogeneous FPGA architectures
- can interface with existing CAD flows:
 - Quartus
 - Modelsim
 - VPR (no heterogeneity)
- Can be used as front end to the following heterogeneous CAD Flow





Heterogeneous FPGA CAD Flow

- Input:
 - HDL design

```
module small (a, b, c, out);  
  input[5:0] a, b, c;  
  output [5:0]out;
```

```
    assign out = ({2'b00,a[2:0]} * b) + (b & ~c));  
endmodule
```






Heterogeneous FPGA CAD Flow

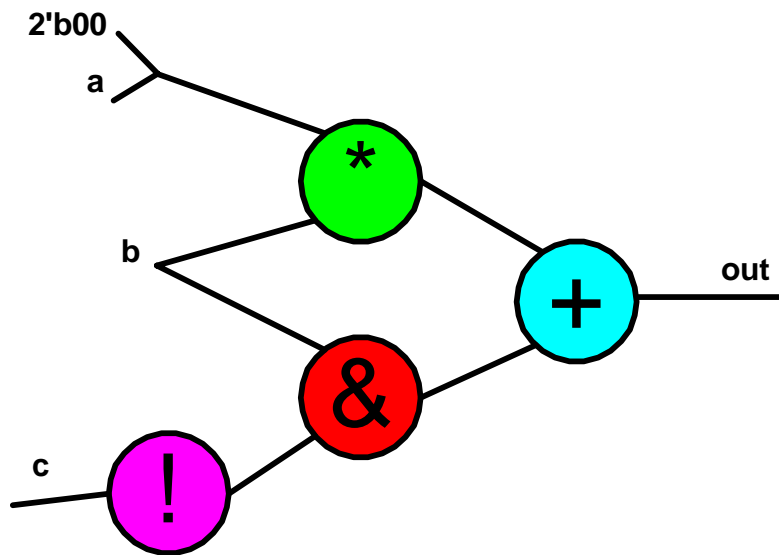
- Parse HDL
 - Icarus creates an intermediate representation

```
MODULE = small
PARAMETERS = {
    {a, input, 5},
    {b, input, 5},
    {c, input, 5},
    {out, output, 5}}
NETS = {{out, {a1}}}
EXPRESSIONS = {
    {e0 = op cat, 2'b00, a},
    {e1 = op*, e0, b},
    {e2 = op~, c},
    {e3 = op&, b, e2},
    {e4 = op+, t1, t3}}
LEFT_ASSIGNMENT = {
    {a1 = e4}}
```



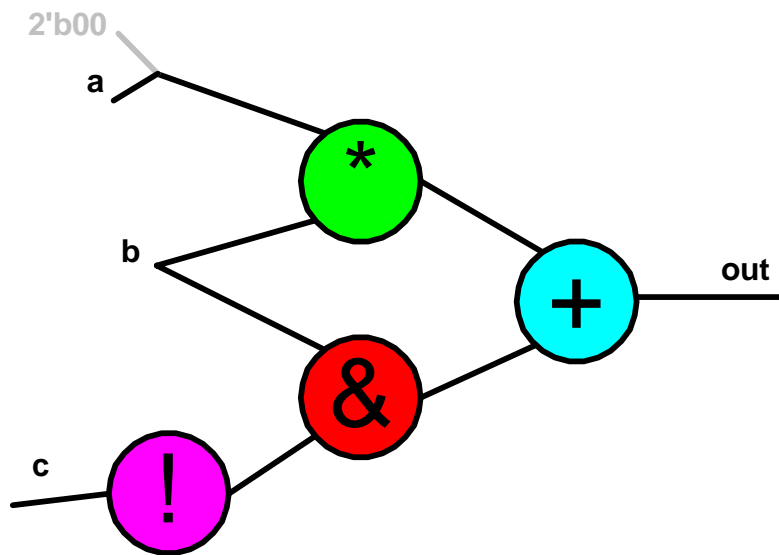
Heterogeneous FPGA CAD Flow

- Elaboration [Odin]
 - Convert into a netlist
 - Preserve high-level Information



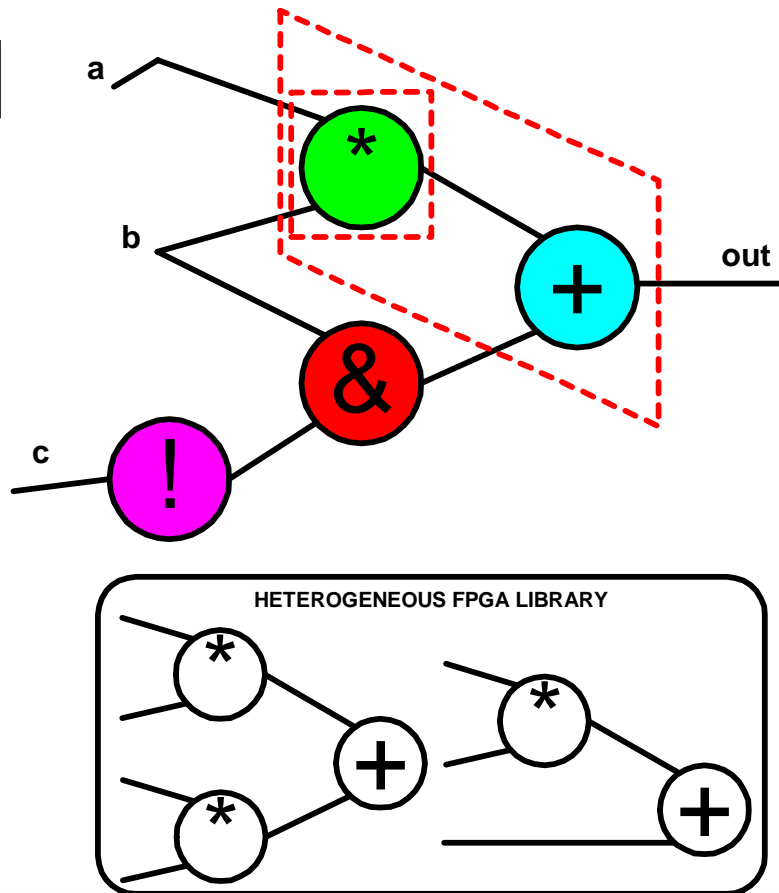
Heterogeneous FPGA CAD Flow

- Optimize RTL [Odin]
 - Arithmetic Operations
 - Finite State Machines
 - Multiplexers



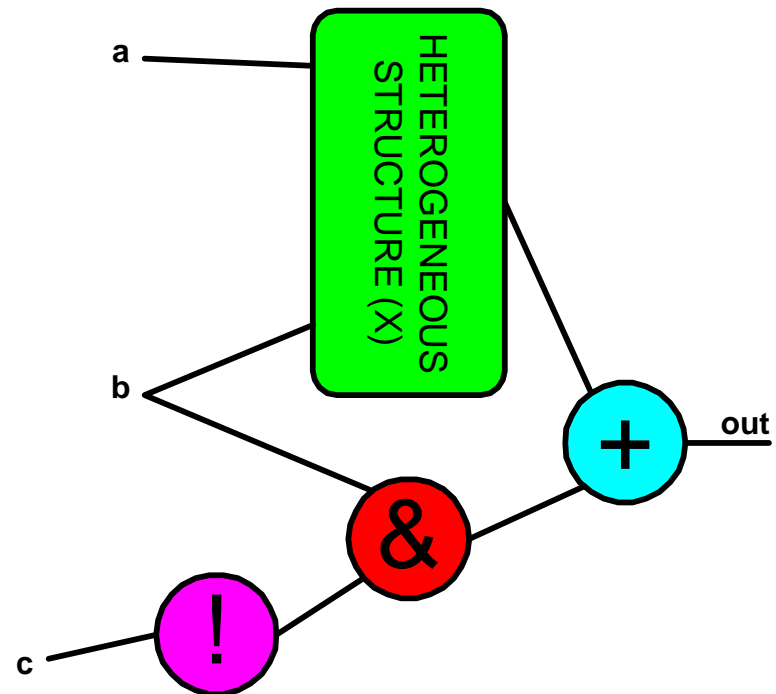
Heterogeneous FPGA CAD Flow

- Partial Mapping [Odin]
 - Identify high-level functions that map into hard structures



Heterogeneous FPGA CAD Flow

- Partial Mapping [Odin]
 - Bind to hard structures or soft fabric





Heterogeneous FPGA CAD Flow

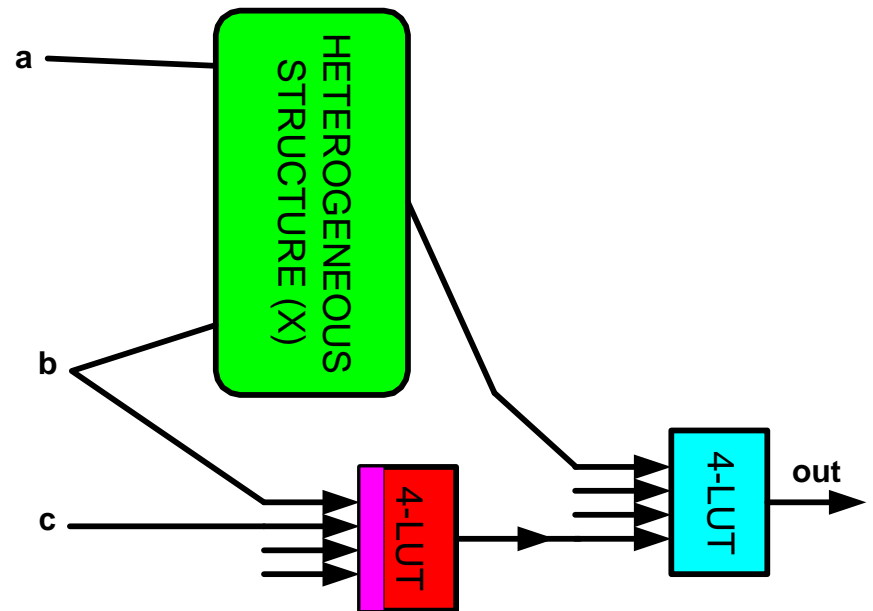
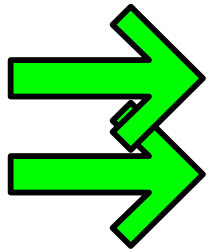
- Output
 - Netlist targeting remainder of heterogeneous FPGA CAD flow

```
module small (a, b, c, out);  
  input[5:0] a, b, c;  
  output [5:0]out;  
  
  not(c, e2);  
  and(b, e2, e3);  
  lpm_mult(a[3:0],b,e1);  
  lpm_add(e1,e3,out);  
endmodule
```



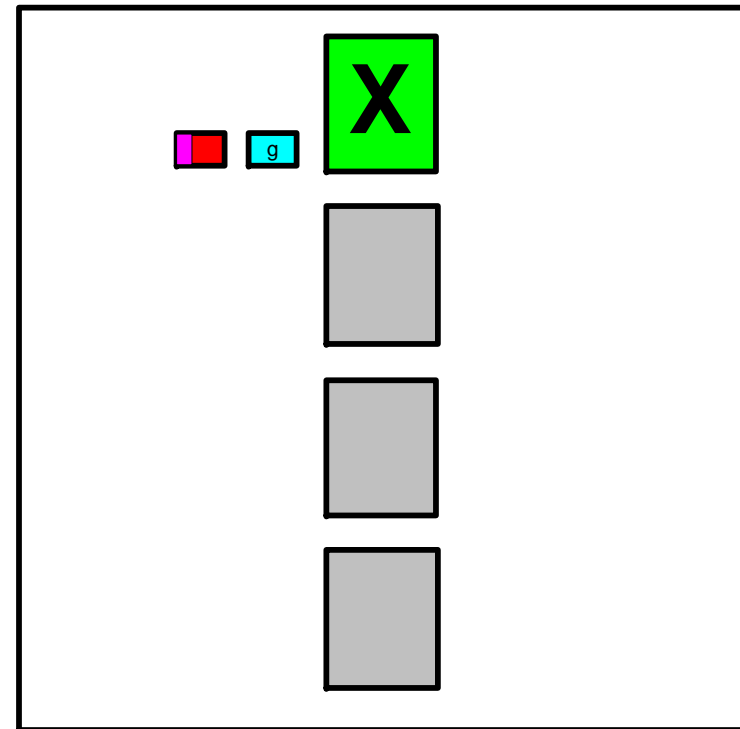
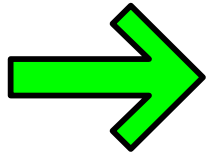
Heterogeneous FPGA CAD Flow

- Input – HDL
- Odin
- Logic Optimize
- Technology Map
- Place
- Route
- Output – FPGA bitstream



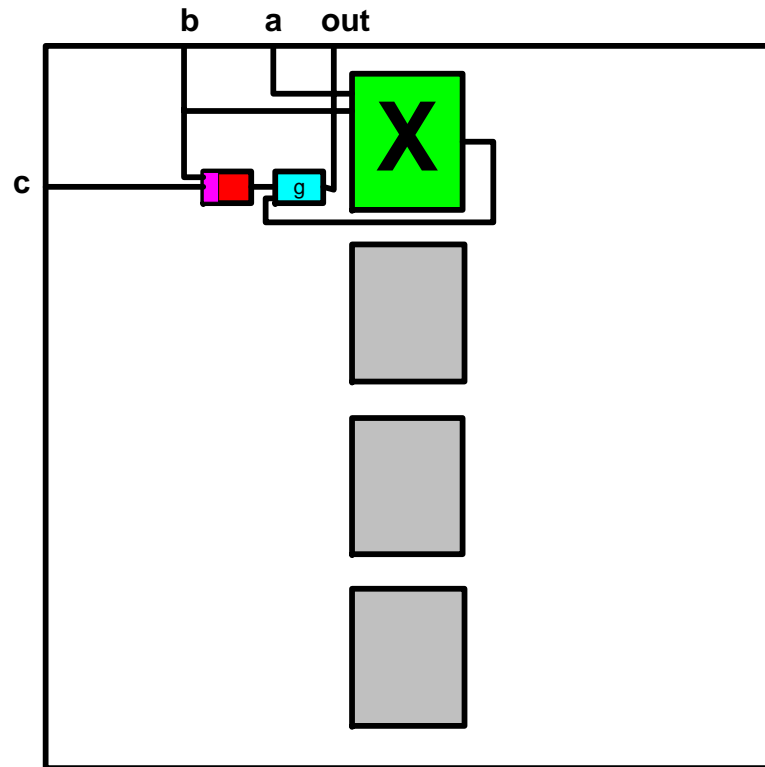
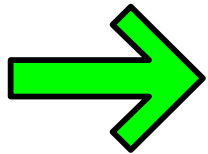
Heterogeneous FPGA CAD Flow

- Input – HDL
- Odin
- Logic Optimize
- Technology Map
- Place
- Route
- Output – FPGA bitstream



Heterogeneous FPGA CAD Flow

- Input – HDL
- Odin
- Logic Optimize
- Technology Map
- Place
- Route
- Output – FPGA bitstream

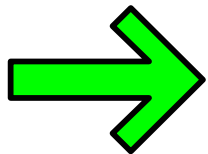




Heterogeneous FPGA CAD Flow

- Input – HDL
- Odin
- Logic Optimize
- Technology Map
- Place
- Route
- Output – FPGA
bitstream

```
10000101010110101000111101010  
10101010101000101010010101001  
11
```





Mapping to Heterogeneity

- Partial mapping maps functionality to hard structures on FPGAs

Step 1: Identification Algorithm

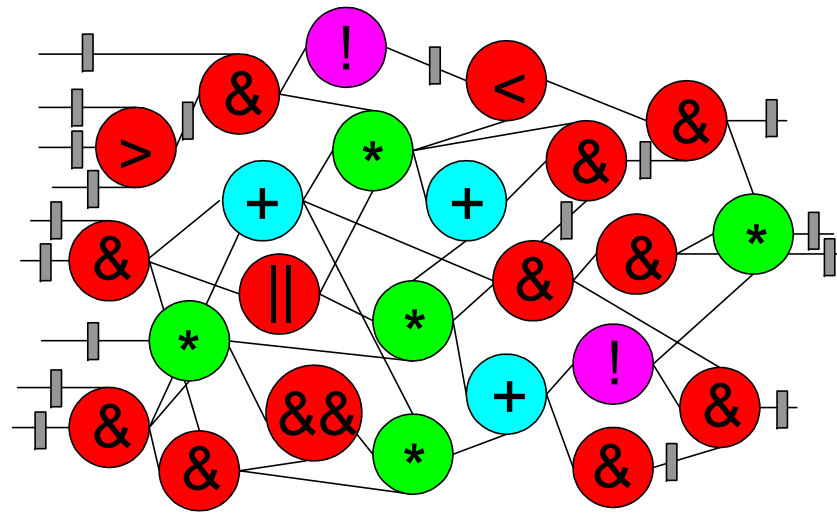
Step 2: Binding Algorithm



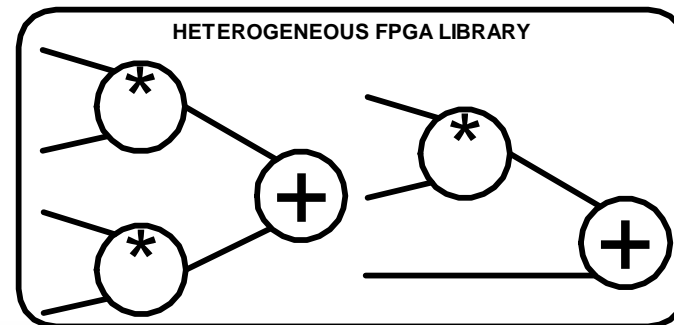
Partial Mapping Identification Step

Input:

1. Netlist:



2. Library
describing hard
structure:





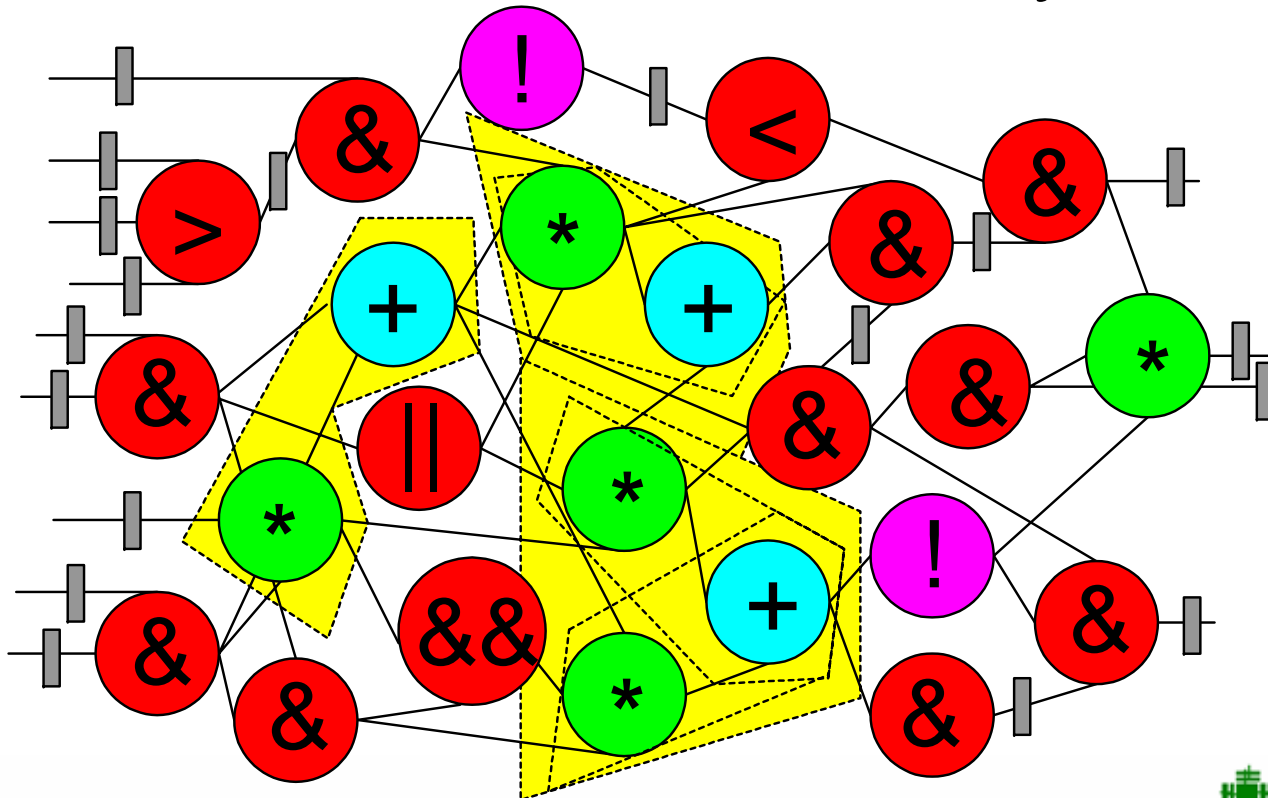
Identification Goal

- Identify all portions of the circuit that could make use of some version of the hard structure on the FPGA



Identification Output

- Netlist with identified functionality



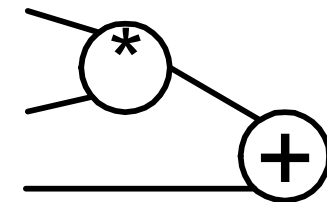
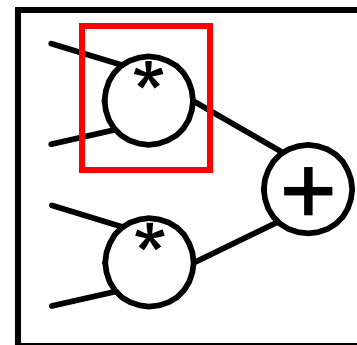
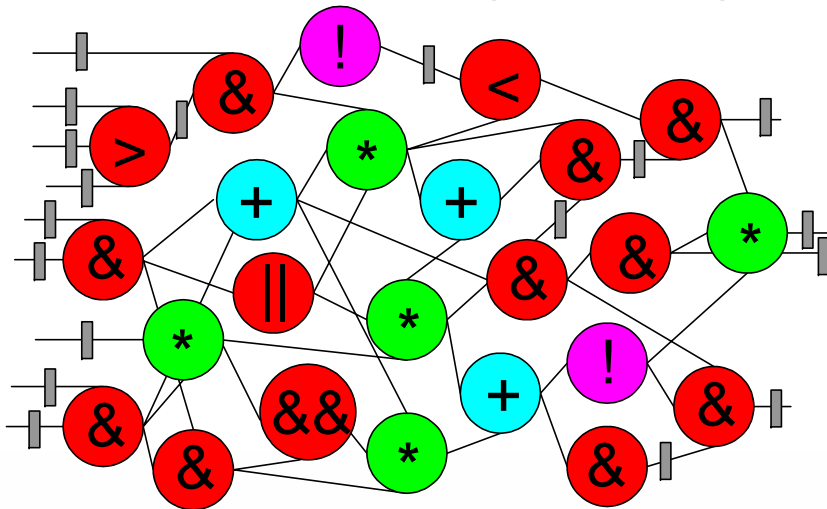
Algorithm

For each (f = function available on hard structure)

$seed$ = a unique part of the f

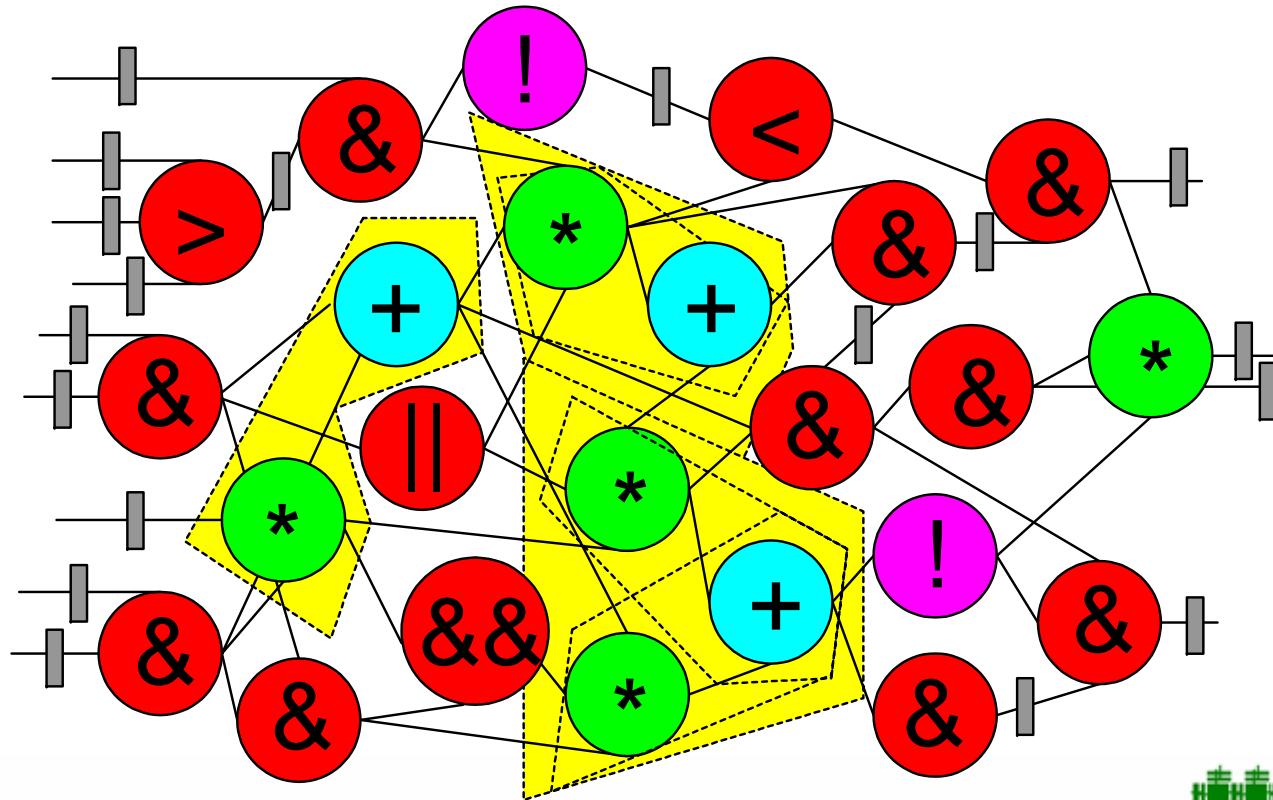
For each (e = element in netlist of type $seed$)

RECORD if ($match(f, e)$)



Partial Mapping Binding

- Input: Netlist with identified functionality





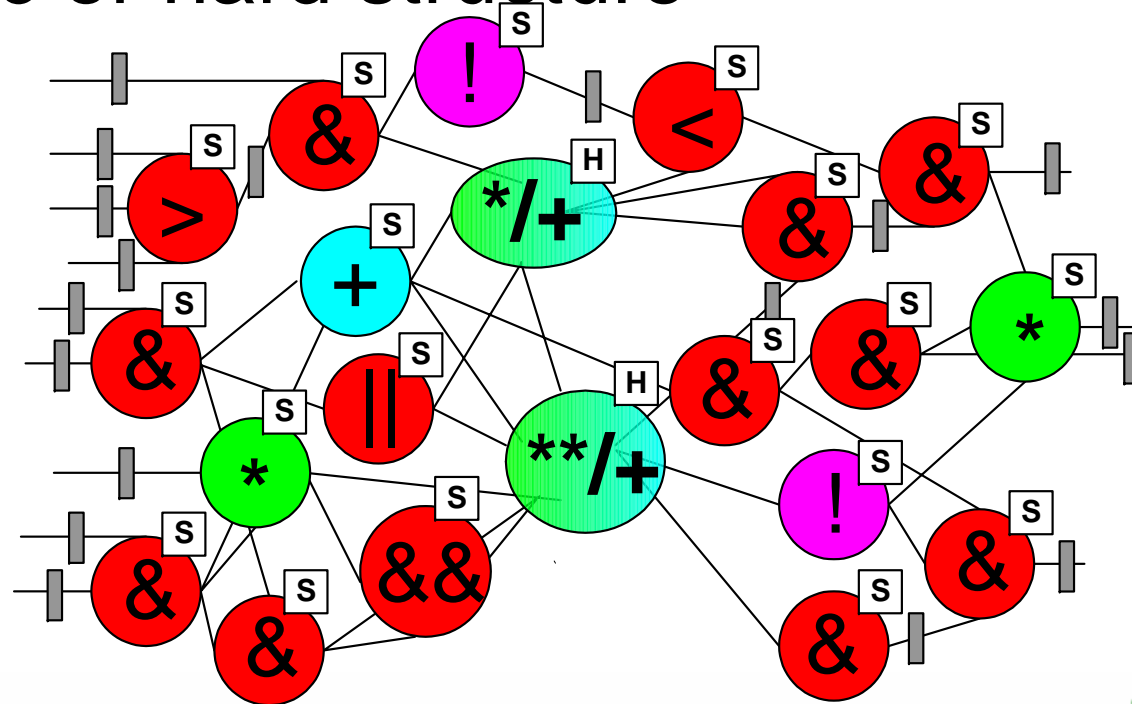
Binding - Goal

- Decide how each element in the netlist will be mapped on the FPGA



Binding - Output

- Netlist with each element bound to soft fabric or hard structure





Binding Algorithm

- Map all identified functionality to FPGA implementations
 - Only find 2 advanced structures in our benchmarks





Mapping Optimizations

- Flip-flops
 - Need to consider the clear signal
- Adder
 - Map to dedicated adder logic on FPGA





RTL Optimizations

- Arithmetic Optimizations
- One-hot Re-encoding of Finite State Machines
- Multiplexer Collapsing



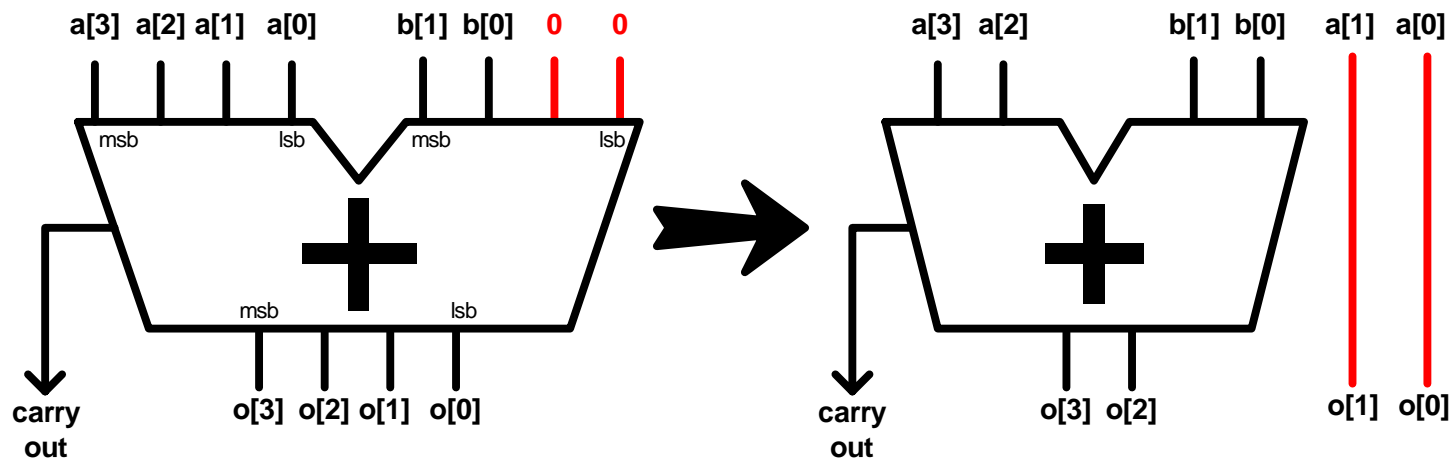


Arithmetic Optimizations

- Goal = Map to smaller arithmetic units
 - Constant propagation
 - Downstream tools don't always do this for arithmetic structures



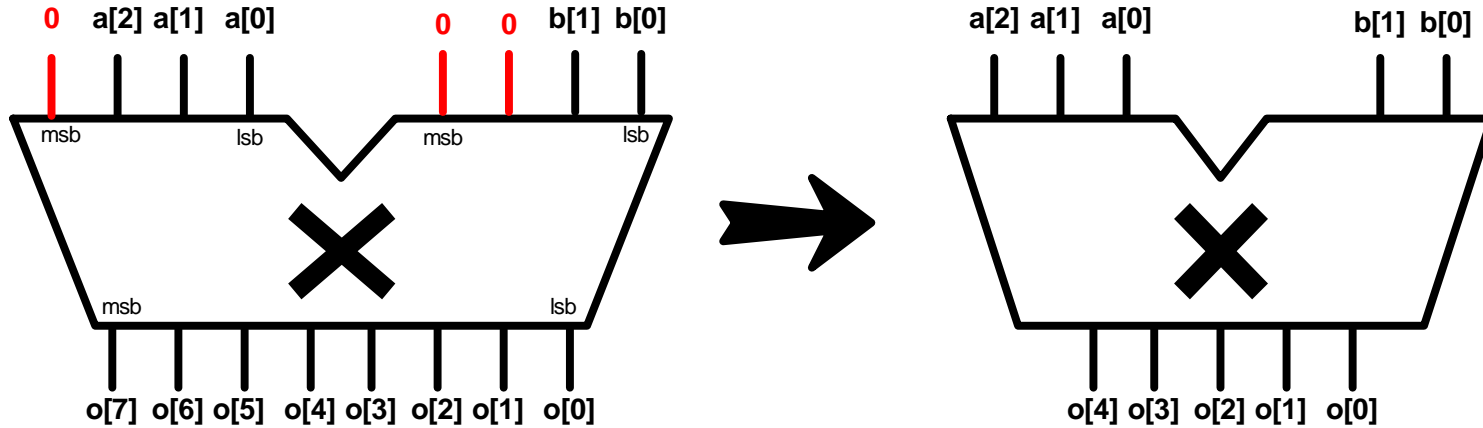
Arithmetic Optimizations - Examples



$$\begin{array}{r} a[3] \ a[2] \ a[1] \ a[0] \\ + \ b[1] \ b[0] \ 0 \ 0 \\ \hline o[3] \ o[2] \ o[1] \ o[0] \end{array}$$

$$\begin{array}{r} a[3] \ a[2] \\ + \ b[1] \ b[0] \\ \hline o[3] \ o[2] \ a[1] \ a[0] \end{array}$$

Arithmetic Optimizations - Examples



$$\begin{array}{r}
 0 \quad a[2] \quad a[1] \quad a[0] \\
 * \quad 0 \quad 0 \quad b[1] \quad b[0] \\
 \hline
 o[7] \quad o[6] \quad o[5] \quad o[4] \quad o[3] \quad o[2] \quad o[1] \quad o[0]
 \end{array}$$

$$\begin{array}{r}
 a[2] \quad a[1] \quad a[0] \\
 * \quad b[1] \quad b[0] \\
 \hline
 o[4] \quad o[3] \quad o[2] \quad o[1] \quad o[0]
 \end{array}$$



One-hot re-encoding of Finite State Machine

- One-hot encoded FSM use less routing and less logic [Golson93]
- Odin identifies and re-encode FSMs



Identifying State Machines in Verilog

```
always @(w or CS)
begin
```

```
  case (CS)
    A :if (w==0) NS = A;
      else NS = B;
    B :if (w==0) NS = B;
      else NS = A;
  endcase
```

```
end
```

```
always @(posedge clock)
```

```
begin
```

```
  CS <= NS;
```

```
end
```

Combinational case statement. CS is the state register

All assignments to state Register are constant or feedback loop



Convert to one-hot

- Once identified change the size of the state register and re-encode constants





Problems with FSM re-encoding

- Not guaranteed to find all state machines





Verification

- Odin has been tested by simulating and verifying results through modelsim
 - cf_cordic
 - fir_scu_rtl
 - molecular dynamics





Quality of Results





Basic Goal

- Show Odin produces comparable results to an industrial front-end synthesis tool





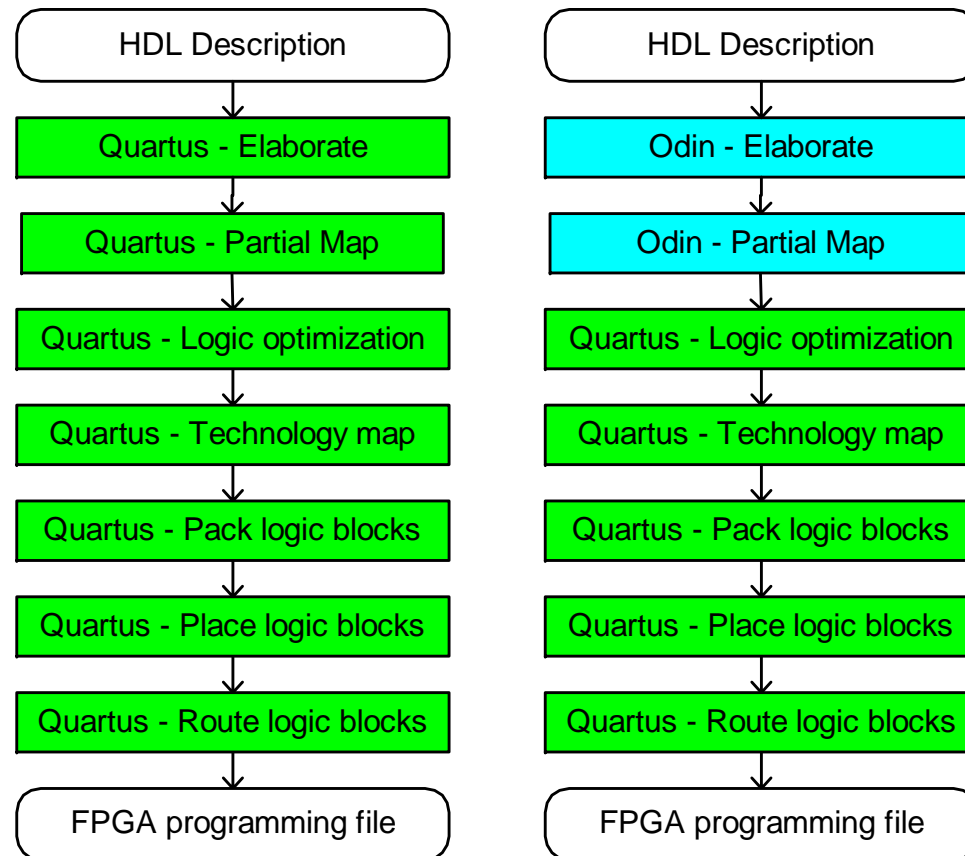
Experimental Setup

- Compare Odin against Altera's Quartus front-end synthesis
 - Use Quartus back-end
- FPGA Used: Stratix I
- CAD Flow based on Altera Quartus 4.1



CAD Flows

- Comparison CAD flows
- Both use Quartus back-end





Benchmarks

- Collection of Verilog Benchmarks
 - Opencores.org
 - SCU-RTL
 - Texas-97
 - Benchmark Suite for Placement-2001
 - Local designs converted from VHDL
 - Raytrace
 - Molecular Dynamics
 - Stereo Vision



Results – Area Comparison

Designs	Ratio	Designs	Ratio
fft_258_6	1.34	reed_sol_decoder1	1.03
iir1	1.73	reed_sol_decoder2	1.09
iir	1.14	md	1.41
fir_3_8_8	1.00	cordic_8_8	1.42
fir_24_16_16	1.00	cordic_18_18	1.45
fir_scu_rtl	0.55	MAC1	0.98
diffeq_f_systemC	1.23	MAC2	0.99
diffeq_paj_convert	0.72	CRC33_D264	1.00
sv_chip1	0.97	des_area	0.88
sv_chip2	1.02	des_perf	0.84
sv_chip2_no_mem	0.98	sv_chip0	1.02
rt_raygentop	1.02	sv_chip0_no_mem	0.98
rt_raygentop_no_mem	1.33	sv_chip3_no_mem	0.79
rt_top	1.14	rt_frambuf_top	1.44
rt_top_no_mem	1.37	rt_frambuf_top_no_mem	1.19
oc45_cpu	1.42	rt_boundtop	1.59

- Ratio < 1 means results from Odin smaller



Results Summary

- Ratio < 1 means results from Odin better
- Geometric Average
 - Area Ratio: **1.11**
 - Speed Ratio: **1.05**





Effectiveness of RTL Optimizations

- Show how techniques improve results generated by Odin





Results – DSP block reduction

- Arithmetic Optimizations cause all improvement
 - 27.8% less DSP-blocks





Results – LE Reduction

Optimization	Percent Improvement
Partial Mapping	3%
Arithmetic optimizations	5%
One hot state reencoding	11%
Multiplexer Collapsing	81%

- Overall 4% improvement





Results – Speed Improvement

Optimization	Percent Improvement
Partial Mapping	13%
Arithmetic optimizations	15%
One hot state reencoding	27%
Multiplexer Collapsing	45%

- Overall 5.6% improvement





Summary

- Odin generates results comparable to Quartus
 - Give numbers
- Showed the relative importance of RTL optimizations





Summary

- Odin available:
 - <http://www.eecg.toronto.edu/~jayar/software/odin/>
 - Open Source
 - GPL software license





Future Work

- Architect better hard structures that are more widely usable

