

C-to-Verilog.com: High-Level Synthesis Using LLVM

Nadav Rotem, Haifa University



Computing tradeoffs

- Different kinds of computational problems
- Different kind of architecture solutions

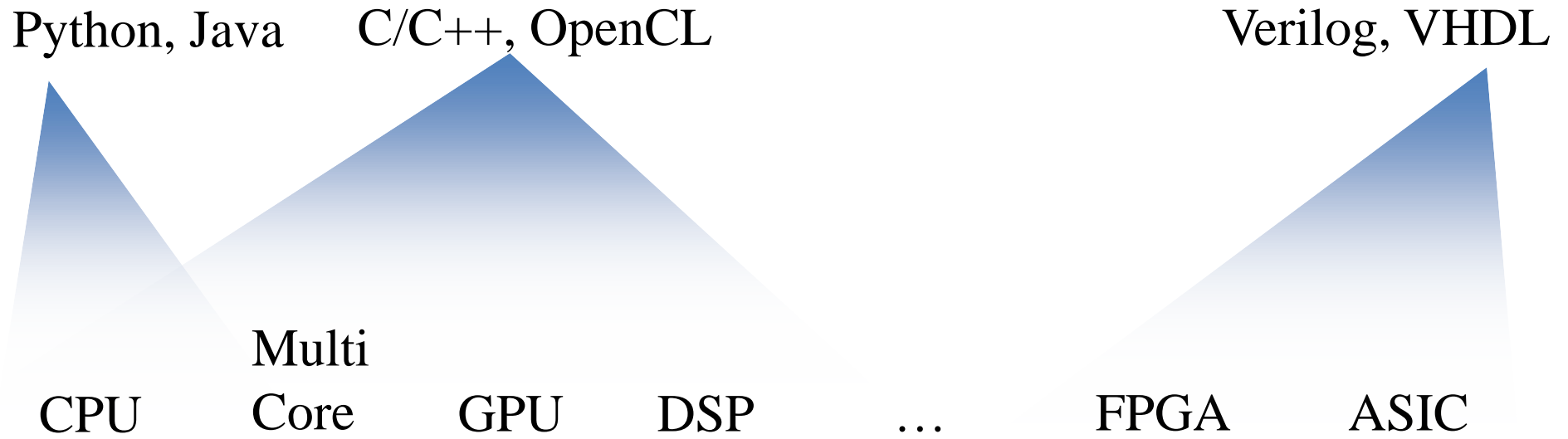
Camera Image
De-noise

GSM
communication

Phonebook
Manager

Render
Graphics





General Purpose

Easy to program.
Flexible. Slow.

Custom

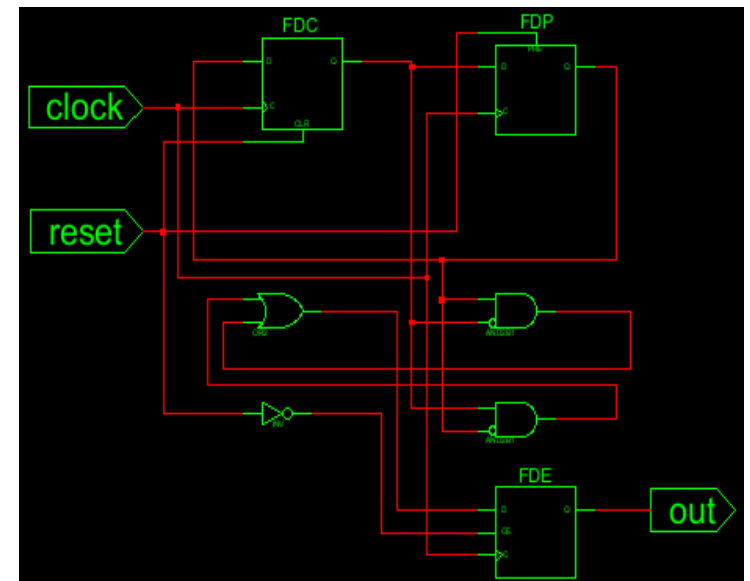
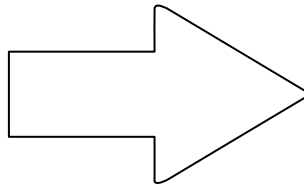
Performance, power efficient. Difficult to program.

Introduction to High- Level Synthesis

Hardware description languages

- Complex digital systems are made of basic logic elements (AND, NOT, FF, etc.)
- Designers use Hardware Description Languages to describe logic blocks
- Use C-like syntax to express hardware

```
module toplevel(clock,reset,out);  
  input clock;  
  input reset;  
  output reg out;  
  reg flop1;  
  reg flop2;  
  always @ (posedge reset or posedge clock)  
  if (reset) begin  
    flop1 <= 0;  
    flop2 <= 1;  
  end else begin  
    flop1 <= flop2;  
    flop2 <= flop1;  
    out <= flop2 ^ flop1;  
  end  
endmodule
```



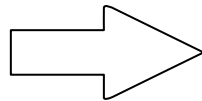
The problem with HDL

- Need to 'think' hardware
 - All parts of the circuit operate at the same time
 - Explicit notion of time (clock, synchronization)
 - Explicit notion of space (size and connectivity of components)
- Extremely long compilation cycle
- Difficult to develop and verify
- Details, Details, Details ...

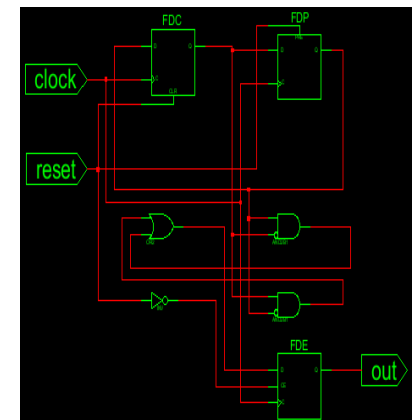
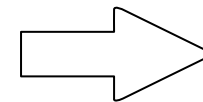
High-level synthesis

- HLS: Compilation of high-level languages, such as C, to Hardware Description Languages.
- Easier to write and test code in C
- Use a subset of C
 - No IO, recursion, jump by value, etc.
 - Standard hardware/software interface

```
while(true) {  
    out = val1 ^ val2;  
    ...  
}
```

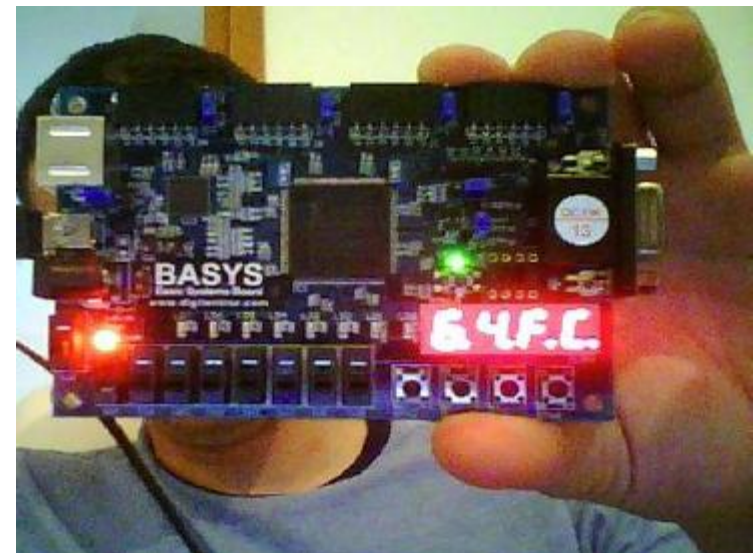


```
module toplevel(clock,reset,out);  
    input clock;  
    input reset;  
    output reg out;  
    reg flop1;  
    reg flop2;  
    always @ (posedge reset or  
posedge clock)  
    if (reset) begin  
        flop1 <= 0; flop2 <= 1;  
    end else begin  
        flop1 <= flop2; flop2 <= flop1;  
        out <= flop2 ^ flop1;  
    end  
end  
endmodule
```



C-to-Verilog.com

- LLVM-based high-level synthesis system
- Developed as a graduate research project
- Website is web-interface for the synthesis system
- Free, Open, etc.

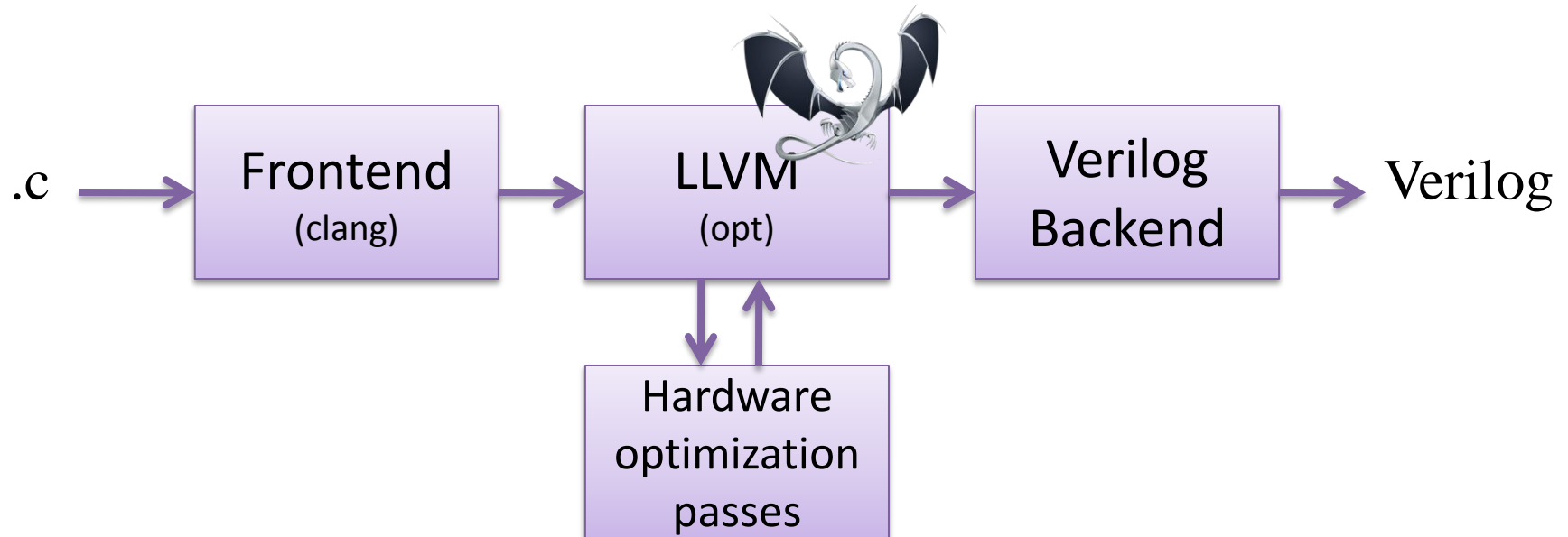


High-Level Synthesis using LLVM



HLS using LLVM

- Use Clang and LLVM to parse and optimize the code
- Special LLVM passes optimize the code at IR level
- HLS backend synthesize the code to Verilog



HLS backend for LLVM

Simple High-Level Synthesis

- It is trivial to compile sequential C-like code to HDL
- A state-machine can represent the original code
- We can create a state for each 'opcode'
- Example:

```
entry:  
    %A = add i32 %B, 5  
    %C = icmp eq i32 %A, 0  
    %br i1 %C, label %next, label %entry  
    ...
```



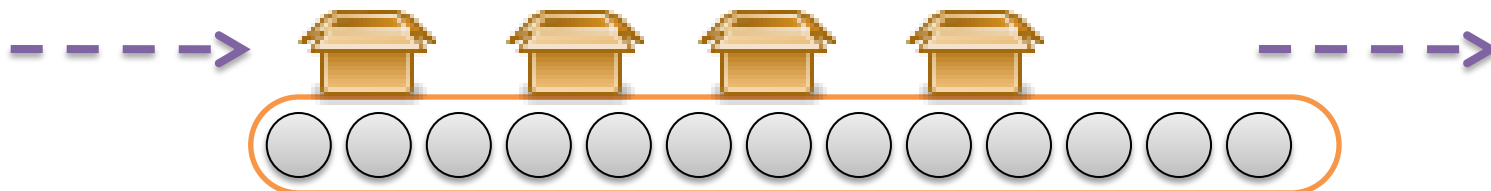
```
case (state)  
    ST0: begin  
        A <= B + 5;  
        state <= ST1;  
    end  
    ST1: begin  
        C <= (A == 0)  
        state <= ST2;  
    end  
    ST2: begin  
        if (C)  
            state <= ST9;  
        else  
            state <= ST0;  
        end  
    ...  
endcase
```

High-Level synthesis challenges

- The simple state-machine translation is inefficient
- We want to optimize:
 - Fast designs (few clock cycles to complete)
 - High-frequency (low clock latency)
 - Size and resource efficient (few gates, memory ports)
 - Low-power

Scheduling pipelined resources

- Generally, in HLS resources can be synthesized
 - Unlimited registers, arithmetic ops, etc.
- Some resources are limited, and need to be shared.
 - External memory ports
 - ASIC Multipliers (for FPGA synthesis)
- Often, hardware resources are 'pipelined', to gain high frequencies.
 - Multiplier – 5 stages, Memory – 2 stages, etc.



List Scheduling

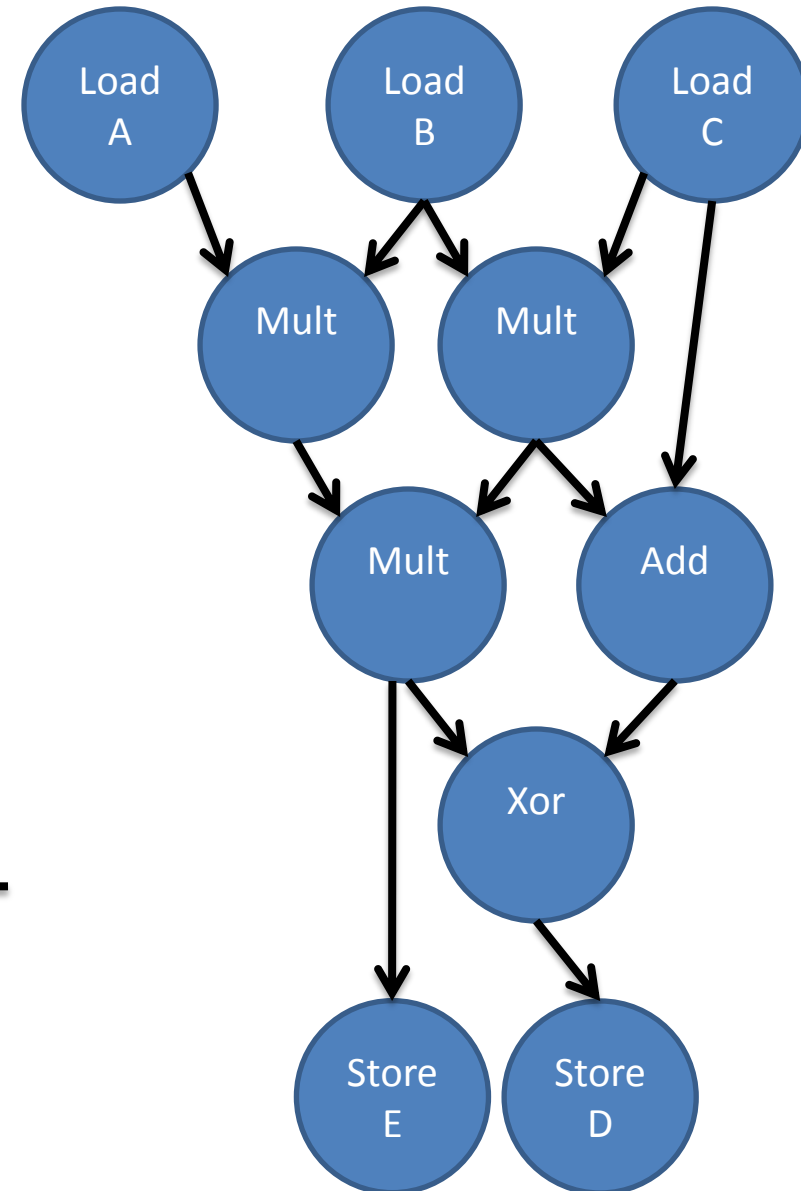
- Schedule a single basic block
- Convert a DDG into a [Time x Resource] table
- Requirements:
 - Preserve DDG dependencies
 - Expose parallelism
 - Conserve resources, use pipelined resources
- After scheduling, HDL syntax generation is simple

Example (bad)

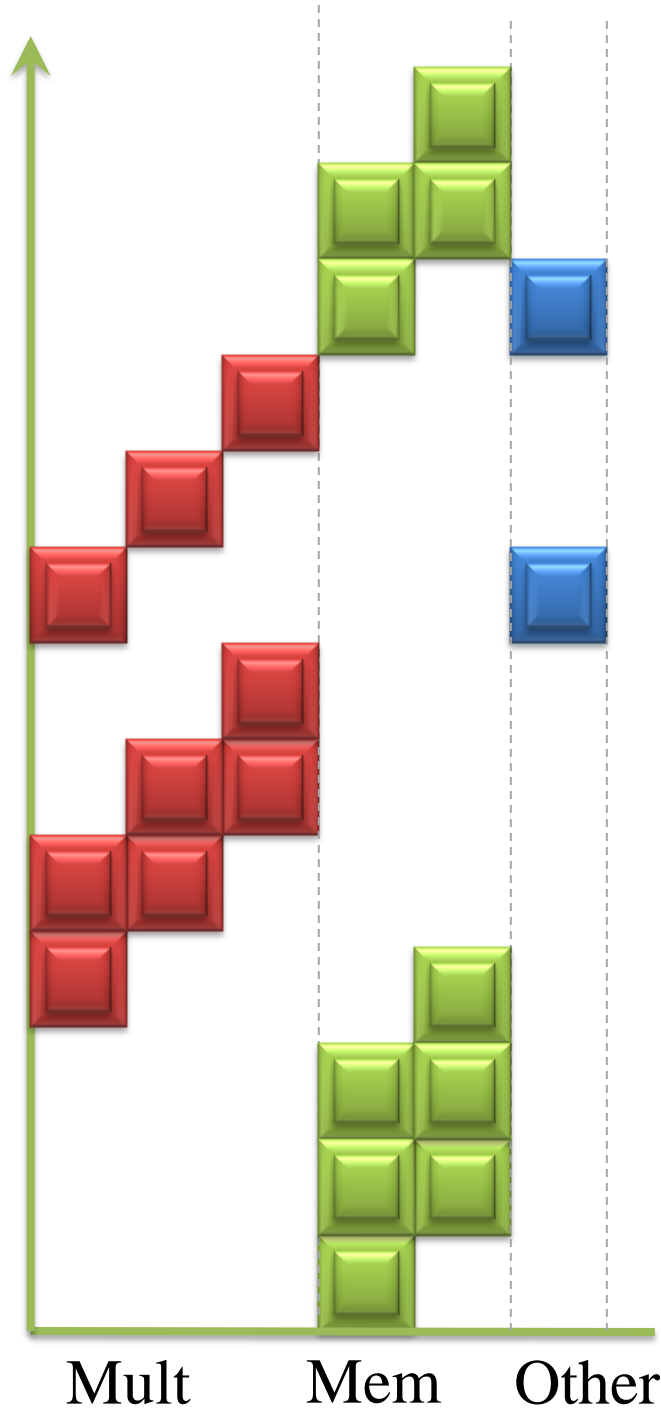
- Multiplier – 3 cycles
- Load/Store – 2 cycles
- Other – 1 cycle

+2
+2
+2
+3
+3
+3
+1
+1
+2
+2

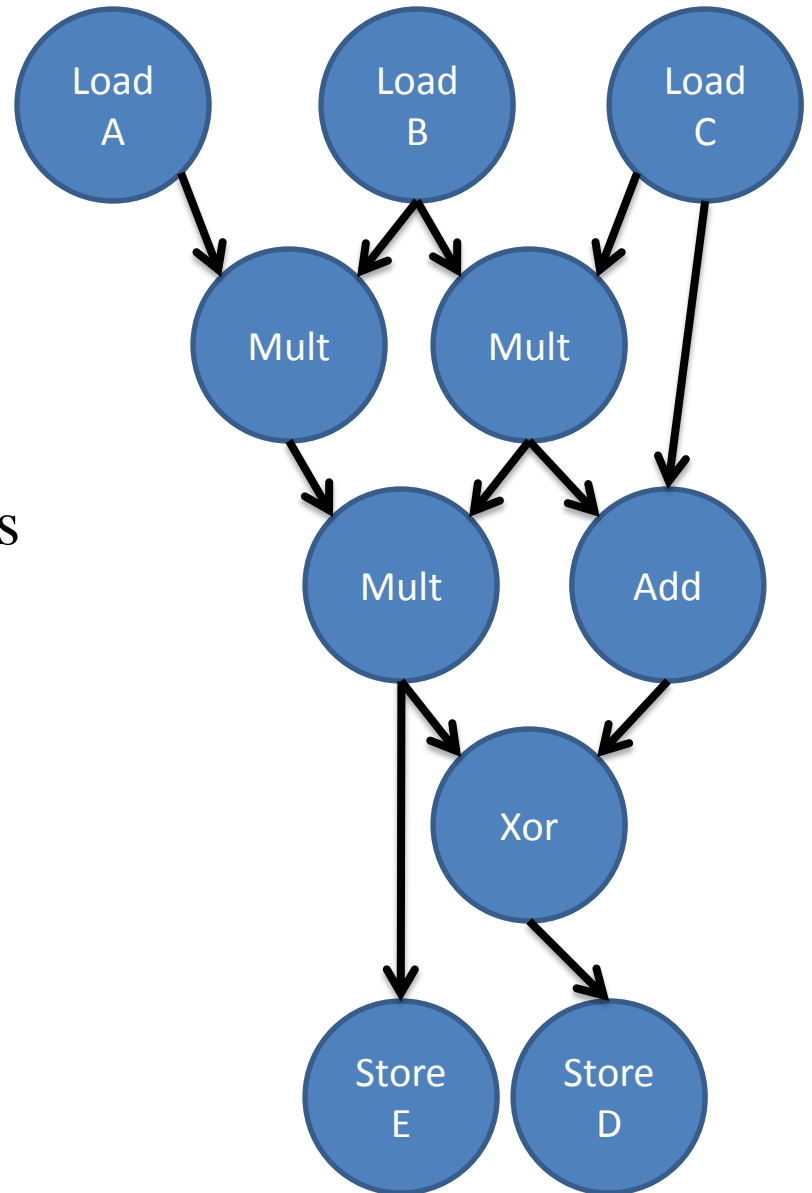
21 cycles



Time



13 cycles



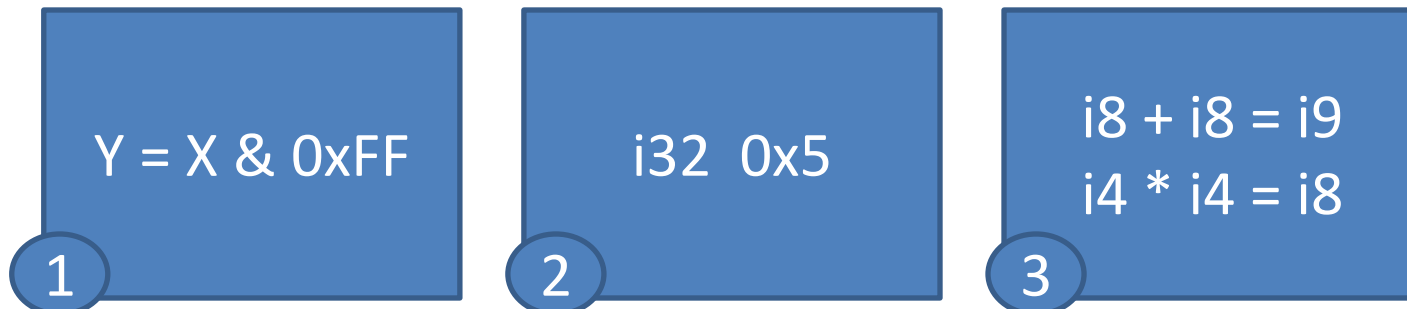
IR Optimizations for hardware

Reduce-bitwidth-opt

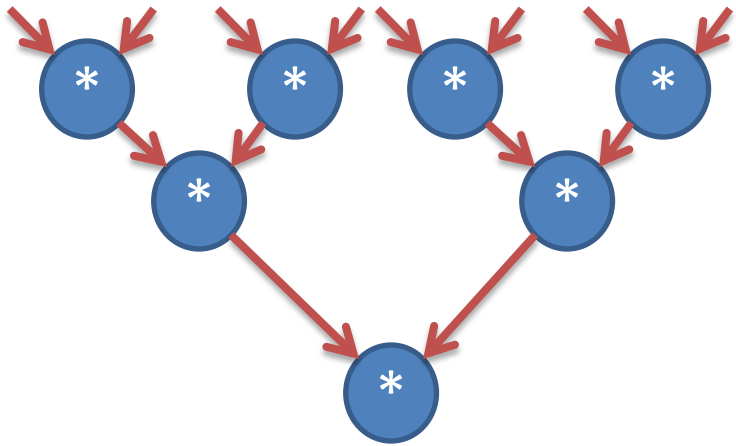
- CPUs have fixed execution units
- In hardware synthesis, arithmetic operations are synthesized into circuits
- Fewer bit width arithmetic operations translate to smaller circuits which operate at higher frequencies

Reduce-bitwidth-opt

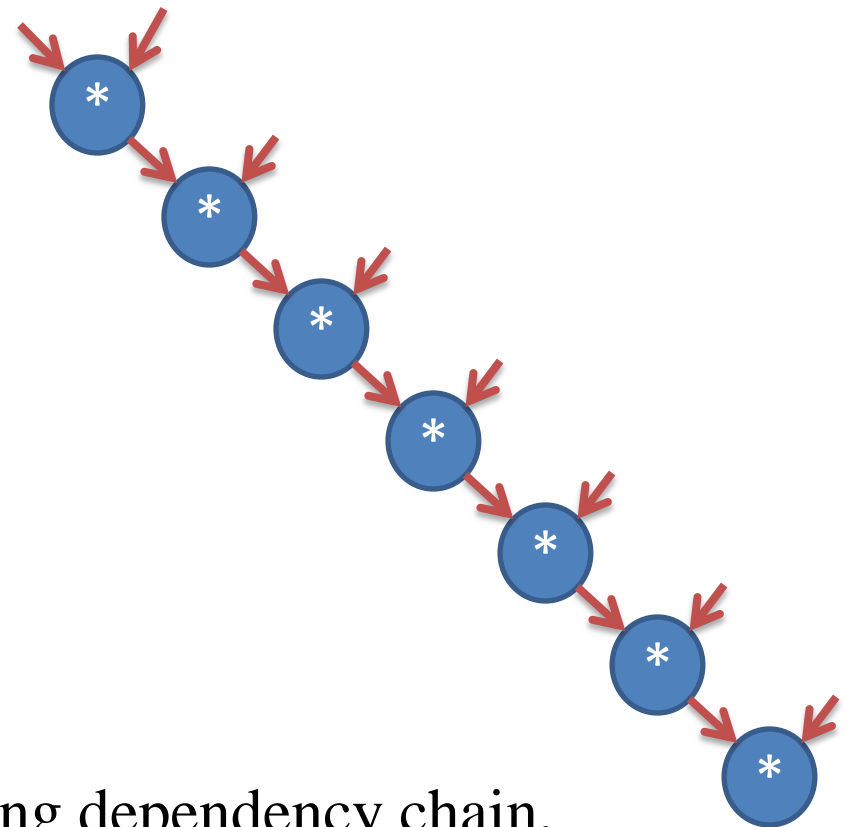
- Reduce bit width in several cases:
 1. Detect local bit reducing patterns (masks)
 2. Reduce constant integers to lowest bit-width
 3. Use smallest possible arithmetic operation based on input width
- Simple LLVM Pass



Arithmetic tree height reduction



Short dependency chain,
High parallelism



Long dependency chain,
Low parallelism

Arithmetic tree height reduction

- Simple LLVM pass
- Collect long chains of arithmetic operations and balance them
- Only for commutative arithmetic and logic operations
- Not suitable for software where number of registers is unlimited

HLS flow example

Pop-count example

```
// count the number of 1's in a word
unsigned popCnt(unsigned input) {
    unsigned sum = 0;
    for (unsigned i = 0; i < 32; i++) {
        sum += (input) & 1;
        input = input/2;
    }
    return sum;
}
```


Pop-count

- Runtime:
 - The program has a loop, which executes 32 times.
- Size:
 - Has several 32-bit registers.
 - Has control-flow logic.
- Frequency:
 - Has 32bit-adders, long carry-chains.
- IR-level optimizations can be very beneficial

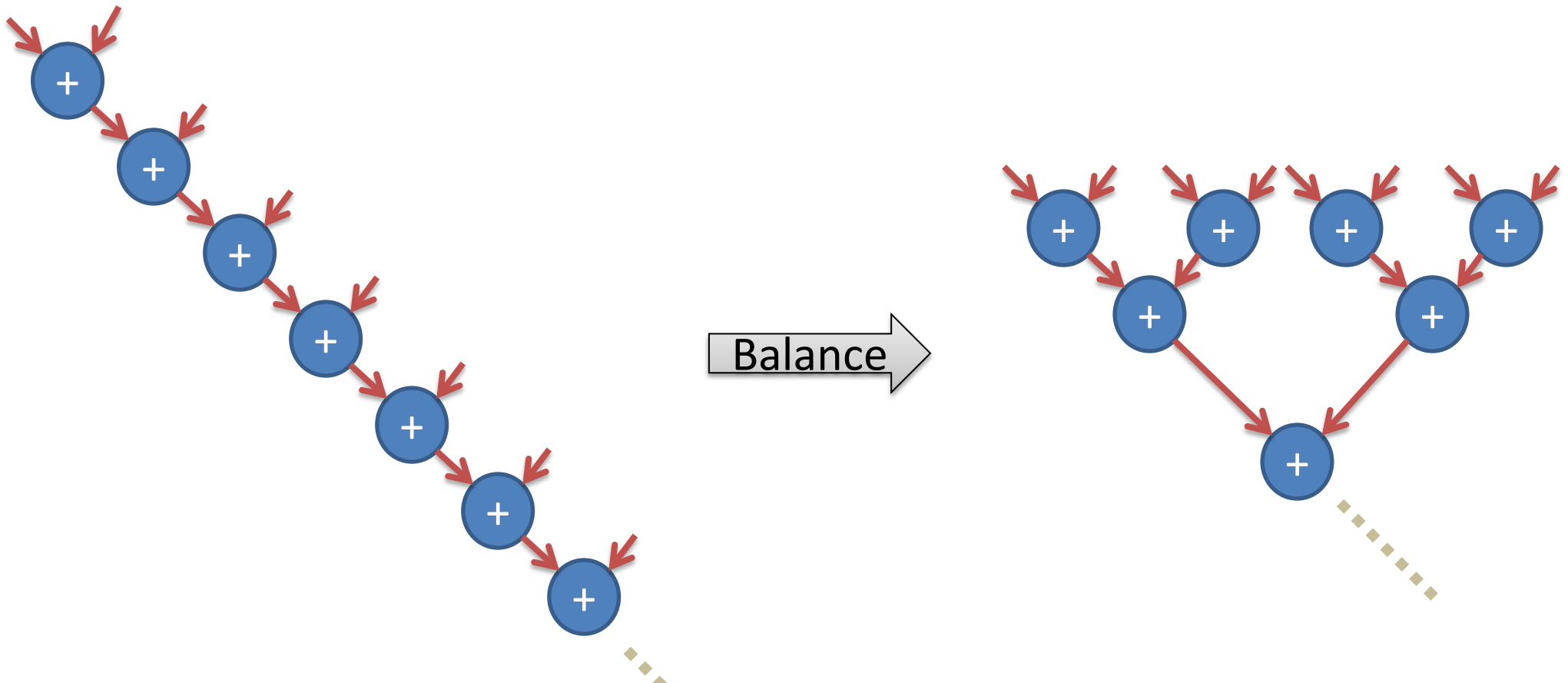
Pop - count

- First, we let LLVM unroll and optimize the loop

```
// count the number of 1's in a word
unsigned popCnt(unsigned input) {
    unsigned sum = 0;
    sum += (input >> 0) & 1;
    sum += (input >> 1) & 1;
    sum += (input >> 2) & 1;
    ...
    ...
    return sum;
}
```

Pop - count

- Next, we balance the long-chain of adders to become a tree of 31-additions

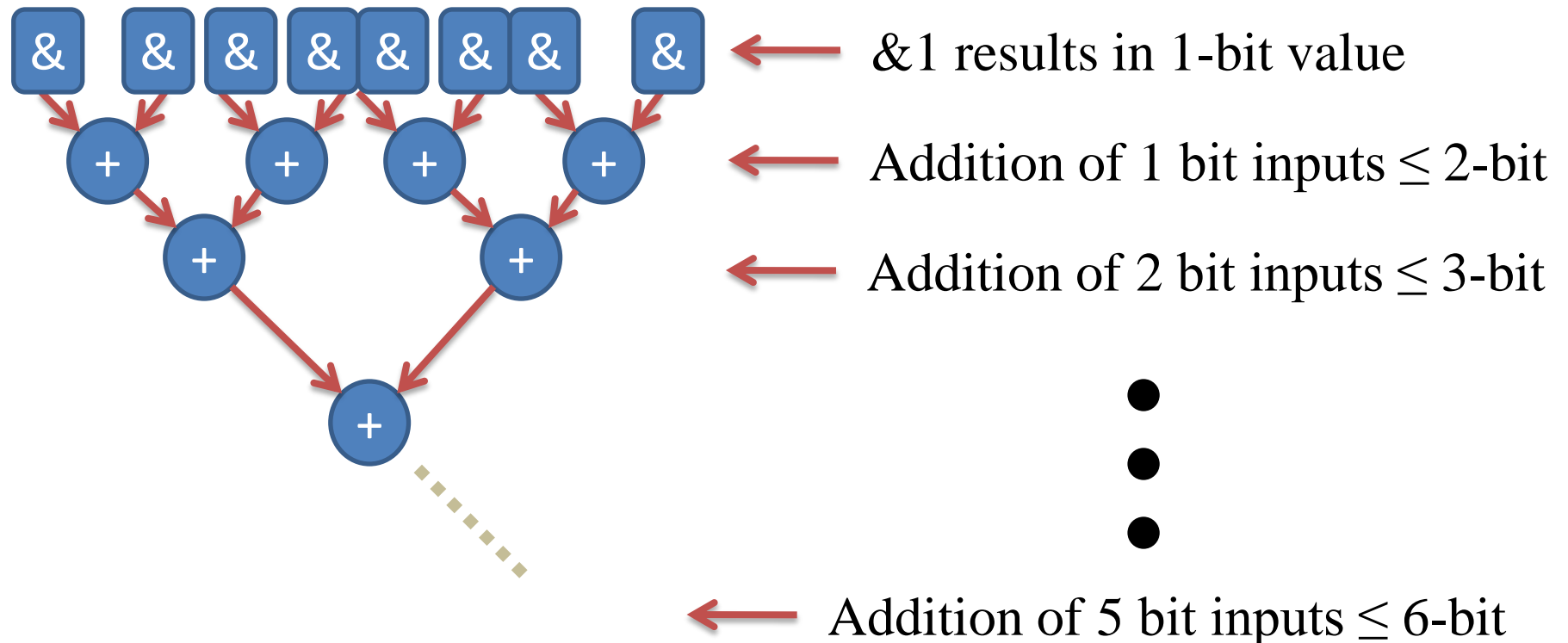


Pop - count

```
// count the number of 1's in a word
unsigned popCnt(unsigned input) {
    unsigned sum0, sum1, sum2, sum3 ... t0, ...
    t0 = (input >> 0) & 1;
    t1 = (input >> 1) & 1;
    t2 = (input >> 2) & 1;
    ...
    ...
    sum0 = t0 + t1;
    sum1 = t2 + t3;
    ...
    sum30 = sum29 + sum 28;
    return sum;
}
```

Pop - count

- Finally, we'll reduce the bit-width of each operation

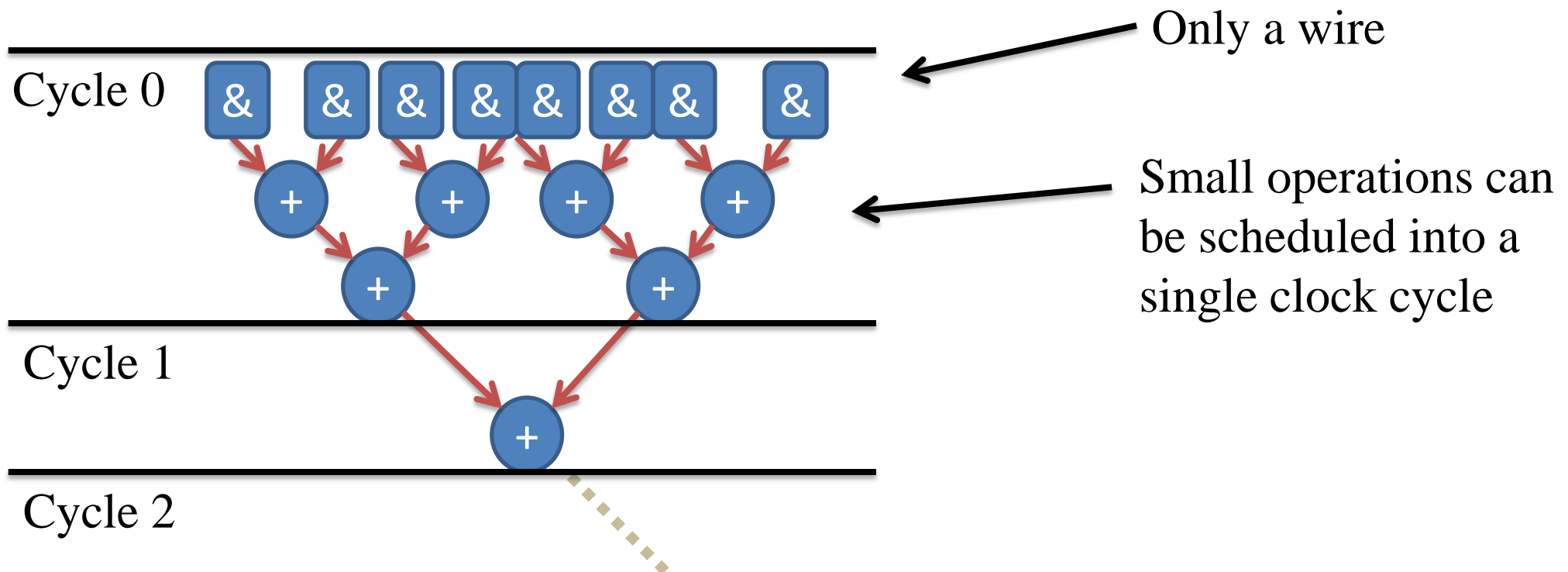


Pop - count

```
// count the number of 1's in a word
unsigned popCnt(unsigned input) {
    uint1_t t0, t1 ...
    uint2_t sum0, sum1, sum2, sum3 ...
    uint3_t sum17, sum18, sum19, sum20 ...
    ...
    uint6_t sum31;
    ...
    t0 = (input >> 0) & 1;
    ...
    sum0 = t0 + t1;
    ...
    sum31 = sum29 + sum 30;
    return sum;
}
```

Pop – count

- Finally, we pass the hw-optimized IR to the backend for scheduling and syntax generation



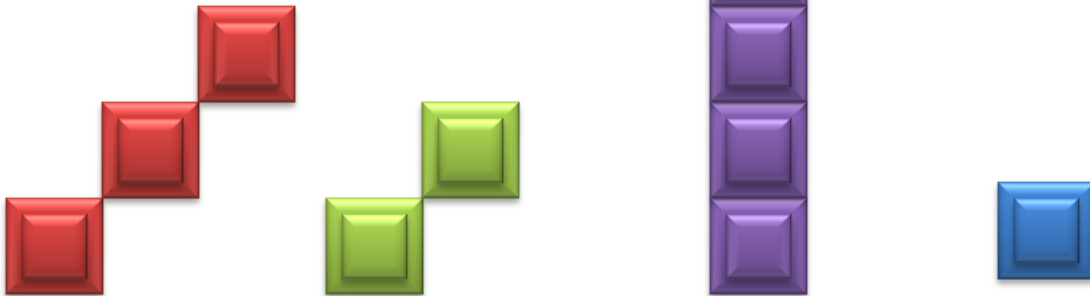
Pop - count

- IR-level optimizations are very beneficial
 - Size: fewer and smaller registers, no control flow
 - Frequency: smaller arithmetic ops (32bit -> 6 bits)
 - Cycles: Fewer cycles (32 -> 4)

Conclusion

- High-level synthesis automates circuit design
- LLVM is an invaluable tool when developing a HLS compiler
- HLS compiler is made of IR-level optimization passes and a scheduling backend

Questions ?



Ref

- [C-to-Verilog.Com](#)
- [High Level Synthesis](#)
- [Synthesis of Pipelined Arithmetic Units](#)
- [HLS Publications](#)