

High level synthesis: From C to RTL

Alexander Wold

August 18, 2025

Overview

- What is high level synthesis (HLS)
- Which tools are available
- Typical applications
- HLS compilation process
- Designing hardware with C
- Compiler transformations
- Non-optimal code for synthesis
- References

What is High level synthesis?

- Generate hardware from C or another high level language.
- Higher abstraction level (behavior).
- Improved productivity:
 - ▶ Faster implementation
 - ▶ Faster verification
 - ▶ Faster time to market
- Multiple hardware implementation alternatives can be generated from one HL implementation.
- A HL model can be used to generate hardware which meet different performance requirements and resource constraints.

High level synthesis tools

- Open source and commercial tools are available:
 - ▶ Catapult C (Siemens)
 - ▶ HDLCoder (MathWorks)
 - ▶ HLS Compiler (Altera)
 - ▶ Vitis Unified IDE (Vivado HLS) (AMD)
 - ▶ LegUp (Microchip Technology/University of Toronto)
 - ▶ Dynamatic (EPFL)

Typical applications

- Streaming or data-flow oriented applications:
 - ▶ Compute-kernels (High performance computing)
 - ▶ Encryption and hashing algorithms
 - ▶ Digital signal processing
 - ★ Video and audio encoding
 - ★ Image processing
 - ★ Seismic data (e.g. oil and gas)
 - ★ Radars

RTL example

- Process computes $(a+b)*c$ in VHDL
- How to pipeline the process?

```
process: add_and_multiply(clk)
begin
  if rising_edge(clk) then
    d<=(a+b)*c;
  end if;
end process;
```

RTL example

- Can we pipeline it more?

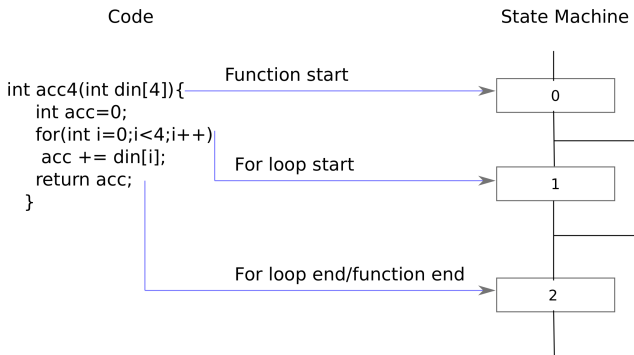
```
process: add_and_multiply (clk)
begin
  if rising_edge (clk) then
    d0<=a+b;
    d<=d0*c;
  end if;
end process;
```

HLS example

```
#pragma HLS PIPELINE
int add_and_multiply(int a, int b, int c) {
    return a+b*c;
}
```

- `#pragma` is a C compiler directive
- method specified by the C standard to provide additional information to the compiler. ¹

¹<https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html>



Code

```
int acc4(int din[4]){  
  int acc=0;  
  for(int i=0;i<4;i++)  
    acc += din[i];  
  return acc;  
}
```

Operations

Write

Write

Read

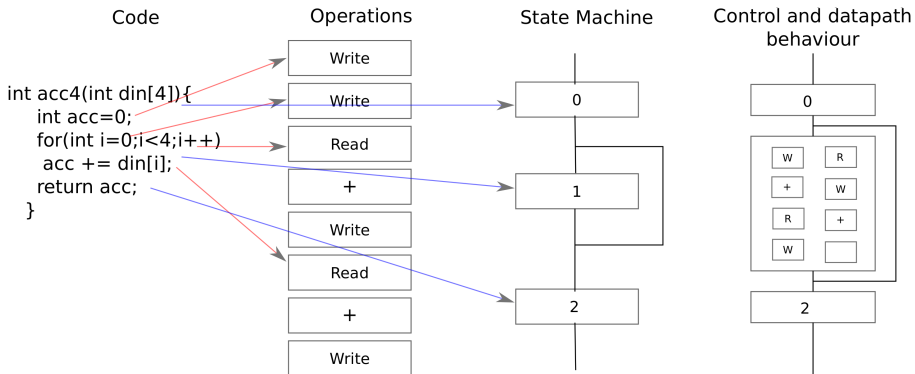
+

Write

Read

+

Write



HLS compilation process

- Data flow graph analysis
- Resource allocation
- Scheduling

Data flow graph analysis

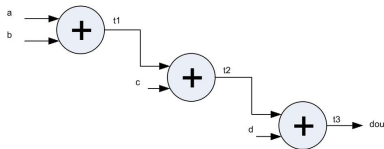
- High level synthesis starts by analyzing data dependencies in the code.
- This leads to a Data Flow Graph (DFG)
- Parts of code without dependencies can be executed in parallel

```
void accumulate(int a, int b, int c, int d, int &dout){  
    int t1,t2;  
    t1 = a + b;  
    t2 = t1 + c;  
    dout = t2 + d;  
}
```

Data flow graph analysis

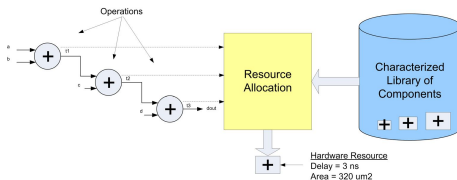
- High level synthesis starts by analyzing data dependencies in the code.
- This leads to a Data Flow Graph (DFG)
- Parts of code without dependencies can be executed in parallel

```
void accumulate(int a, int b, int c, int d, int &dout){  
    int t1,t2;  
    t1 = a + b;  
    t2 = t1 + c;  
    dout = t2 + d;  
}
```



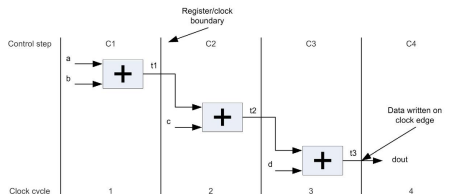
Resource allocation

- Based on the assembled DFG, each operation is mapped onto a hardware resource.
- This process is called resource allocation.
- The implementation is annotated with both timing and area information. This is used during scheduling.



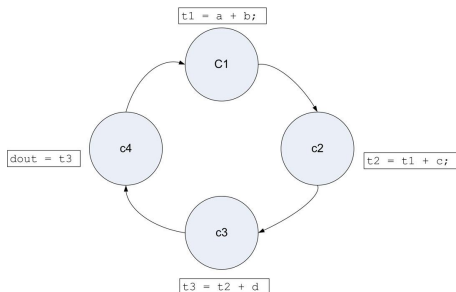
Scheduling

- HLS adds time to the design during the scheduling.
- Scheduling takes the DFG operations and decides when they are performed.
- Registers are added based on the target clock frequency.



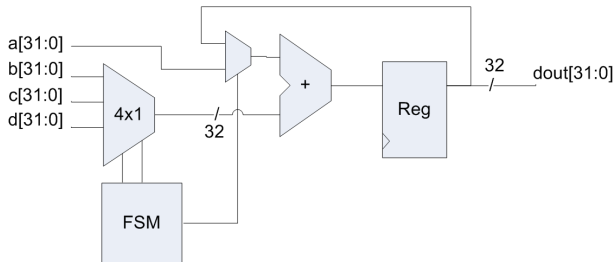
Scheduling

- A data path state machine (DPFSM) is created to control the scheduled design.
- In this design, four states are required to execute the schedule.
- These states are referred to as control steps or c-steps.



Scheduling

- The resulting hardware generated from the schedule will vary depending on the design constraints.
- Design constraints include resource allocation (size) and performance (performance).



Demo

Not all code is suited for synthesis!

- Need to keep in mind that the code ends up as hardware.
- Not all algorithms are suited for hardware implementations.
- Control logic see little benefit.

HLS restrictions

- Restrictions in the HLS flow often requires rewriting C functions.
- C code targeting HLS is therefore less portable.
- Different HLS tools synthesis C code differently further decreasing portability.
- Programming code often require HLS tool specific adaptations.

Arbitrary precision data types

- Register sizes are defined on CPU (8/16/32/64 bits)
- With HLS we can use arbitrary precision data types (e.g. 4 bits)
 - ▶ GNU GMP² is not supported
 - ▶ Arbitrary Precision apint.h in Vitis
 - ▶ Algorithmic C³ (Siemens)

```
#include "ap_int.h"

//typedef char dinA_t;

typedef ap_int<6> dinA_t;
```

²<https://gmplib.org/>

³https://github.com/hlslibs/ac_types

System calls and packages

- *stdio.h* functions such as *printf* or *cout*.
- *math.h* *sin*, *cos* and most math functions are not synthesisable.
 - ▶ Vitis implements many *math.h* functions
- Assembly code is not synthesisable.
- System calls are generally not synthesisable.

Recursive functions

- Recursive functions are functions which calls themselves.
- Used to write compact and efficient C code.
- Recursive functions executed on a CPU makes heavy use of the return stack, and are often unbounded.
- Typically a complete rewrite is required in order to get a iterative implementation which can be synthesised.

Function pointers

- Function pointers are usually not supported.
- Needs to be changed to explicit function calls.

```
int main() {  
    void (*fp)(int);  
    fp = func;  
    fp(2);    // function pointer call  
    func(2); // explicit function call  
}  
  
void func(int arg) {  
    printf("%d\n", arg);  
}
```

Dynamic Memory Allocation

- Dynamic memory allocation is typically done with *malloc*
- Dynamic memory allocation is not synthesisable, and we need to use static memory allocations.

```
int static_mem[32];  
int *dyn_mem = malloc(32, sizeof(int));
```

Unbounded Loops

- Loops without finite bounds.
- When the start value, stop value and the increment is constant and defined, the loop is bounded.
- A loop is not bounded when *start*, *stop* or *increment* is passed as a function parameter!

```
void unbounded_loop(int loop) {  
    int i;  
    for ( i = loop; i >= 0; i--)  
        statement;  
}
```

Other restrictions

- Global variables for sharing data between functions are not supported.
- Problematic to pass pointers when using a CPU with MMU.

```
int glob_var=5430;

void func_0() {
    statement glob_var;
}

void func_1() {
    statement glob_var;
}
```

Designing hardware with C

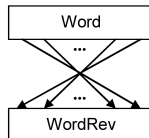
- We need to know how the tools transform C code into hardware in order to get efficient implementations.
- HLS tools perform code transformations on the programming code when generating hardware implementations.
- Code transformations on different levels:
 - ▶ Bit-level
 - ▶ Instruction-level
 - ▶ Loop-level
 - ▶ Data-oriented

Bit-level transformations: bit reversing

- Software (a) implementation of bit reversing compared to a hardware implementation (b).

```
int Reverse(int Word) {  
    int WordRev = 0;  
    for(int i=0; i<32; i++) {  
        WordRev |= (Word & 1);  
        WordRev << 1;  
        Word >> 1;  
    }  
    return WordRev;  
}
```

(a)



(b)

Bit-level transformations: bit-width narrowing

- Variables are often defined with a greater dynamic range than needed.

Consider:

```
for (int i=0; i<4; i++)  
  statements
```

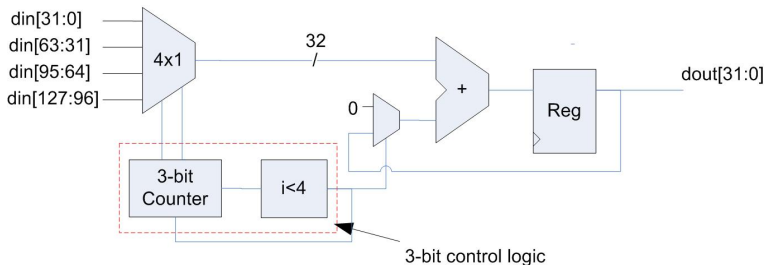
- On a CPU we use predetermined register widths.
- When implemented in hardware we allocate physical resources to the i variable.
- Bit-width narrowing can be determined by static analysis or profiling.

Bit-level transformations: bit-width narrowing

Consider the following example where bit-width narrowing is used to optimize the counter:

```
void accumulate4(int din[4], int &dout){  
    int acc=0;  
    for(int i=0;i<4;i++)  
        acc += din[i];  
    dout = acc;  
}
```

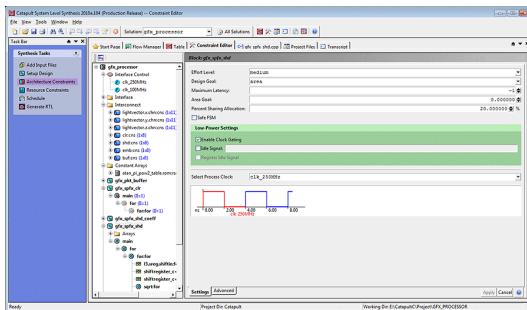
Which results in the following hardware:



Loop-level transformations

- HLS tools generate non-unrolled loops, partial unrolled loops, or fully unrolled loops out of a single implementation.

```
void accumulate4(int din[4], int &dout){  
    int acc=0;  
    for(int i=0;i<4;i++) acc += din[i];  
    dout = acc;  
}
```

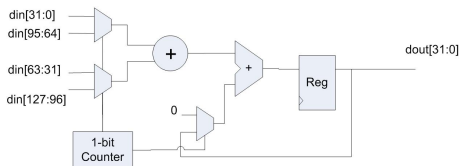


Loop-level trans.: Partial Loop Unrolling

In order to increase throughput consider the example:

```
void accumulate(int din[4], int &dout){  
    int acc=0;  
    for(int i=0;i<4;i+=2){  
        acc += din[i];  
        acc += din[i+1];  
    }  
    dout = acc;  
}
```

Which results in the following hardware implementation:

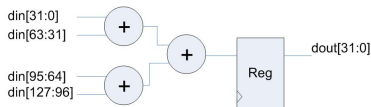


Loop-level trans.: Fully Unrolled Loop

Further increasing the throughput, consider the example:

```
void accumulate(int din[4], int &dout){  
    int acc=0;  
    acc += din[0];  
    acc += din[1];  
    acc += din[2];  
    acc += din[3];  
    dout = acc;  
}
```

Results in a *balanced adder tree*:

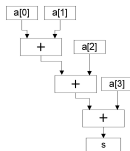


Instruction-level trans: Tree Height Reduction

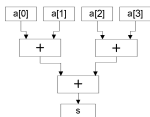
- Reduce the height of a tree of operations by reordering them without changing the functionality.
- THR is applied to (b) which results in the tree shown in (c).
- (d) and (e) shows the scheduling when the data is stored in memory

$$s = a[0] + a[1] + a[2] + a[3]$$

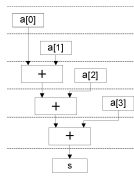
(a)



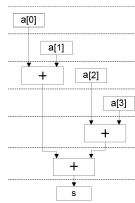
(b)



(c)



(d)



(e)

Operation Strength Reduction (OSR)

- Replace an operation by a computationally less expensive one
- or a sequence of operations

Example

$$2 \ll 1 == 2 * 2$$

- ▶ The multiplication is replaced with a simple shift. The shift only requires changes to the interconnections.

Data-Oriented Transformations

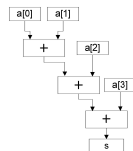
- Data-oriented transformations makes changes to the organization of data structures.
- Common transformations include:
 - ▶ Data distribution
 - ▶ Data replication

Data distribution transformation

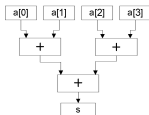
- Partitions the data into many distinct internal memory units or modules.
- Increases throughput.
- Concurrent access.

$$s = a[0] + a[1] + a[2] + a[3]$$

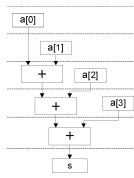
(a)



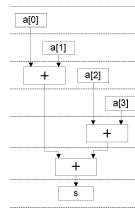
(b)



(c)



(d)



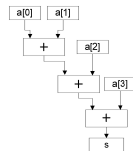
(e)

Data duplication transformation

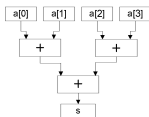
- Increases throughput.
- Concurrent access.
- Consistency issues when modifying data.
- Require more resources.

$$s = a[0] + a[1] + a[2] + a[3]$$

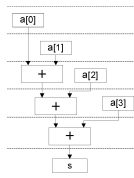
(a)



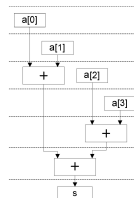
(b)



(c)



(d)



(e)

Other transformations

- Function inlining

- ▶ Dedicated for each function invocation.

```
void accumulate() {  
    accumulate4(din, dout);  
    accumulate4(din2, dout2);  
}  
  
void accumulate4(int din[4], int &dout){  
    int acc=0;  
    for(int i=0;i<4;i++) acc += din[i];  
    dout = acc;  
}
```

Other transformations

- Function outlining
 - ▶ Increases resource sharing.
 - ▶ Reducing parallelism.

References

- Mentor Graphics The High Level Synthesis Blue Book
- Compiling for Reconfigurable Computing: A survey, Cardoso, Diniz, Weinhardt. DOI 10.1145/1749603.1749604
- <https://docs.amd.com/r/en-US/ug1399-vitis-hls/Introduction>
- <https://www.youtube.com/watch?v=hZ2RGwLmXc0>
- https://www.youtube.com/watch?v=5lYq8_bqAcl