High Level Synthesis

Eunike, Pierri, Matthew
Seminar Overview

Significance of HLS

Eunike
- Overview
- What’s so good about it
- What are the challenges it faces

Breakdown of HLS

Pierri
- How it works

Possibilities of HLS

Matthew
- The future of HLS
Introduction to HLS
Software vs. Hardware

ONE SPEEDY BOI
WHAT IS HIGH-LEVEL SYNTHESIS?

“[a design process which enables] the automatic synthesis of high level, untimed or partially timed specifications, such as C or SystemC, to low level cycle-accurate RTL specifications for efficient implementation in ASICs or FPGAs” *

BENEFITS OF HLS

General Perspective
- Decreases code complexity
- Codesign and coverification

Software Perspective
“RTL programming in VHDL or Verilog is unacceptable to most software application developers...”

BENEFITS OF HLS

General Perspective
- Decreases code complexity
- Codesign and coverification

Software Perspective
- Don’t need hardware expertise
- Can benefit from hardware performance

Hardware Perspective
- Can design faster
- Can experiment with hardware faster
DOWNFALLS OF HLS

Design Specifications

- Timing, interface information and constraints need to be specified
- Cannot be implemented on different targets

Choice of Language

- Lack of built-in constructs eg. bit accuracy specification, timing, concurrency...
- Complex constructs eg. pointers, dynamic memory management, polymorphism...
- Too many options in the past
HLS: How it Works
Stages

Parsing & Optimisation

- Transform C, C++ code into an intermediate representation (IR)
- Can take advantage of existing tools, e.g. gcc

Scheduling

- Sort the operations of the IR into a series of control steps
- Can be optimised for minimum resources or time
- Available resource/time constraints can be specified

Binding

- Choose the hardware to be used for each operation (library components, muxes, etc.)
- Introduce registers where values are used across cycles
Choose the hardware to be used for each operation (library components, muxes, etc.)

Introduce registers where values are used across cycles

Transform C, C++ code into an intermediate representation (IR)

Can take advantage of existing tools, e.g. gcc

Sort the operations of the IR into a series of control steps

Can be optimised for minimum resources or time

Available resource/time constraints can be specified

Typical IR is a control & data flow graph (CDFG)

Each node represents a simple operation, e.g. add, read/write, compare

Parsing and optimisation of high-level code can be done using existing tools like gcc

Besides the usual optimisation techniques, some HLS-specific optimisations can be used

\[ \text{out} = (A+B) \times (B-C); \]

Goal: Transform high-level code (C, C++) into IR
**Optimisations**

- **Constant propagation/dead code elimination**
  - Typical compiler technique - avoid recalculation of constant values at run-time

```java
int a = 30;
int b = 9 - (a / 5)
int c = b * 4;
if (c > 10) {
    c -= 10;
}
return c * (60 / a);
```

```java
int c = 12;
if (true) {
    c = 2;
}
return c * 2;
```

```java
return 4;
```
Loop unrolling & pipelining
- Unrolling is typical - write out iterations manually to reduce branching
- On an FPGA we can also execute multiple iterations simultaneously
- Pipelining is done by starting a new loop iteration as soon as data dependencies are cleared, even if the previous one is still in progress
- May even be able to use the same components, depending on the datapath

If-conversion
- Better than branch prediction - execute both branches in parallel, and discard the incorrect one’s results
- Can provide nearly zero-cost branches in some situations
Strength reduction/simplification
- Replace operators with less expensive equivalents
- May also use more specific operators if available, e.g. add $\rightarrow$ increment

\[
\text{res} = x \mod (2^n) \quad \text{or} \quad \text{res} = x \& (2^n - 1);
\]

Range analysis
- FPGA datapath width can be freely changed, unlike processors with a fixed bus size
- Track range of values through a program to minimise bit width of variables and operators

0..4
0..3
ADD
??
**Bitwise analysis**
- Variant of range analysis using bitwise checks
- Performed together with range analysis, as results are better in some cases and worse in others

- The **LegUp** HLS tool also performs profiling-based range analysis, where actual runtime values are recorded and bit-widths are adjusted based on that data
• Memory analysis
  ○ Identify opportunities for parallelism in memory accesses, e.g. writing an array
  ○ May involve splitting an array across multiple memory banks to allow simultaneous access
  ○ **Array scalarization** can be applied to remove a memory access altogether
  ○ Instead of instantiating a memory component for an array, convert it to a list of registers

```c
for (i = 0; i < 4; i++) {
}
```

○ The above example saves a read & write cycle per iteration, and all 4 iterations can be performed at once on the right

```c
A0 = A0 + x;
A1 = A1 + x;
A2 = A2 + x;
A3 = A3 + x;
```
Goal: Organise the CDFG into a series of control steps

- Identify opportunities for parallelism in memory accesses, e.g. writing an array
- May involve splitting an array across multiple memory banks to allow simultaneous access
- **Array scalarization** can be applied to remove a memory access altogether
- Instead of instantiating a memory component for an array, convert it to a list of registers
- Each operation is assigned a control step, which typically corresponds to a single clock cycle

Each of the control steps will eventually become a state in a **finite state machine**, which is the final RTL output of the HLS process

- Time and resource constraints can be specified (e.g. function $f$ must finish within 4 cycles, using at most 2 adders and 1 multiplier)
  - The above example saves a read & write cycle per iteration, and all 4 iterations can be performed at once on the right

```c
for (i = 0; i < 4; i++) {
}
```

```c
A0 = A0 + x;
A1 = A1 + x;
A2 = A2 + x;
A3 = A3 + x;
```
A fully organised CDFG is a **schedule**, and many schedules are possible for each CDFG.

Computing one is an NP-complete problem - many algorithms have been developed based on heuristics to find optimal results.
ASAP (As Soon As Possible)

- From first to last operation, inserts into the earliest control step
- To schedule a new operation, its predecessors must have been scheduled in an earlier step

ALAP (As Late As Possible)

- Opposite of ASAP, starts at final operation and inserts into the latest control step
- Requires successors to have been scheduled in a later step

Both of the above finish successfully if all operations have been scheduled. Both assume infinite resources (i.e. no resource constraints, only time)
Example (4 cycle time constraint):

ALAP: 2 less multipliers, 1 more adder
**FDS** (Force Directed Scheduling)

- Combines *ASAP* and *ALAP* to maximise resource utilization, and therefore minimise total resources required.
- First calculate both *ASAP* and *ALAP*. Any operations that have the same step in both can remain unchanged.
- The remaining ones could potentially be scheduled anywhere between their *ASAP* location and *ALAP* location.
- This difference in steps is called the *range*.
- Working with one type of operation at a time, try each possible control step, calculating the cost function each time to find the minimum.
- The cost function is probability-based and takes into account the expected operations that will be required in each step.
- Scheduling an operator can cause the cost function to change due to data dependencies.
List Scheduling

- Unlike the previous time-constrained algorithms, \( LS \) is resource-constrained
- Working 1 control step at a time, \( LS \) schedules as many operations as possible, subject to data dependencies and resource constraints
- If multiple operations are competing for a resource, one is chosen based on a priority function
- This function is typically its ASAP/ALAP range, where operations with smaller ranges are given higher priority
Other algorithms

- **Simulated annealing**
  - Assists in finding global optima in the presence of local optima
  - Choose control step placements randomly, then calculate some score for the schedule
  - If the score is improved, perform the placement. If not, perform it anyway with a probability that decreases over the life of the algorithm.

- **Genetic algorithm**
  - Also avoids being trapped in local optima thanks to its random mutations
  - Can use starting times of operations as genes
  - Fitness can include time/resource cost, as well as a penalty for non-valid solutions
  - Non-valid solutions are not discarded, as they may provide a path to an optimal valid one
Example: Array addition in Vivado HLS

```c
void bignum_add(uint32_t *a, uint32_t *b, uint32_t *c)
{
    uint32_t tmp;
    int carry = 0;
    int i;

    for (i = 0; i < 256; ++i) {
        tmp = a[i] + b[i] + carry;
        carry = (tmp > 0xFF);
        c[i] = (tmp & 0xFF);
    }
}
```
Example: Array addition in Vivado HLS
Goal: Binds operation to hardware components

```c
void bignum_add(uint32_t *a, uint32_t *b, uint32_t *c)
{
    uint32_t tmp;
    int carry = 0;
    int i;
    for (i = 0; i < 256; ++i) {
        tmp = a[i] + b[i] + carry;
        carry = (tmp > 0xFF);
        c[i] = (tmp & 0xFF);
    }
}
```

- Components are chosen from a chip/board-specific library
- Introduces registers for values used across control steps
- Often combined with the scheduling step in modern tools
- Allows better decision making when scheduling by taking into account different types of available hardware
- For example, choose between memory with multiple ports vs single port, low latency

Scheduling

- Example: Array addition in Vivado HLS
- Goal: Binds operation to hardware components
- Components are chosen from a chip/board-specific library
- Introduces registers for values used across control steps
- Often combined with the scheduling step in modern tools
- Allows better decision making when scheduling by taking into account different types of available hardware
- For example, choose between memory with multiple ports vs single port, low latency
Possibilities of HLS
Goals / Purpose

What should/could HLS offer to a developer/designer?
Goals / Purpose

- Increased design productivity
- Decreased necessity for specific knowledge, increased portability of knowledge across applications & systems
- Better optimisation?
  - System specific quirks
  - Application specific quirks
  - RTL level design
“The Design Productivity Gap refers to a faster increase in the complexity of systems than in the productivity of system designers. In order to solve this problem, the world of Electronic Design Automation is currently evolving towards higher levels of architecture abstraction.”

Pelcat et al.
Design Productivity of a High Level Synthesis Compiler versus HDL. 2016 International Conference on Embedded Computer Systems
Goals / Purpose

- Increased design productivity
- Decreased necessity for specific knowledge, increased portability of knowledge across applications & systems
- Better optimisation?
  - System specific quirks
  - Application specific quirks
  - RTL level design
The Lego Part

Always a trend in decreasing design complexity

- Electronics design
- Computation Hardware design
- Software design
The Lego Part
The Lego Part

Can FPGA’s become a lego part, is this helpful?

Consider a comparison with Microcontrollers
- Hardware itself is not application-specific
- Why would you add an FPGA to help out the microcontroller, how common is this?
The Lego Part

“Graphical processing units (GPUs) offer higher floating point throughput, a favorable architecture for data parallelism and higher memory bandwidth than processors. The systems using GPU-based accelerators however, are inefficient in terms of power consumption.”

Muslim et al.
So why doesn’t HLS make this super easy?
So why doesn’t HLS make this super easy?

- Tool-specific knowledge
- Timing and memory management  
  (information not included in software)
- Optimisation naturally doesn’t want to play with generalised tools
Solutions?

- Tool specific knowledge
- Information not included in base code
Are we barking up the wrong tree?

- VHDL and Verilog in some aspects really step outside of HDL
- What are the strengths and weaknesses of a description in C or C-like languages?
- Can we develop other abstract ways to describe hardware with less shortcomings?
A few relatively small projects exist that try high level description.

- DFiant
  - Scala-based HDL, compiles to RTL

- Clash
  - Haskell-based HDL, compiles to VHDL
Functional Hardware Description

- Portability
- Agnostic
- Middle ground

“Emerging HLSs still fail to deliver a clean separation between functionality and implementation that can yield portable code, while providing general purpose HDL construct.”

Port & Etsion

DFiant: A Dataflow Hardware Description Language
Functional Hardware Description

```haskell
import Data.Bits (xor)
import Data.Vector (fromList, toList)
import Data.Vector.Generic (zipWith, foldl', foldr', map, foldl)

-- | The prefix sum structure to use
prefixSum :: KnownNat n => Vec n GenProp
prefixSum = fromList [GenProp (c `xor` p) | c <- fromList [0..], p <- fromList [0..]]

-- | Carry in
-- | Sum input 1
-- | Sum input 2
-- | (Intermediate carrys, sum outputs)

-- | The carry lookahead adder
carryLookaheadAdder (cIn :: BitVector n) (x :: BitVector n) (y :: BitVector n) = (pack $ reverse carrys, pack $ reverse sums)

-- | The generate properties
-- | The generate properties are the result of applying the specified function to the prefix sums of x and y
-- | The function func is defined as the exclusive or of x and y
-- | The generate properties are then mapped over the prefix sums using the specified function
-- | The result is then packed into a vector

-- | The carry lookahead adder
-- | The carry lookahead adder is defined as the exclusive or of the carry in and the prefix sum of x and y
-- | The carry lookahead adder is then mapped over the generated properties using the specified function
-- | The result is then packed into a vector
```