Lecture 1: Introduction to High-Level Synthesis

CR11 – Hardware Compilation and Simulation
ENS-Lyon – M2IF

Christophe Alias
1. Introduction to High-level synthesis: classical HLS flow and open problems

2. Data transfers: architectures, models and optimizations

3. Front-end: automatic parallelization

4. Back-end: control and channel synthesis
Books

High-Level Synthesis
from Algorithm to Digital Circuit

Compilation Techniques for Reconfigurable Architectures

LOOP TILING FOR PARALLELISM

by Jingling Xue
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder_vhdl_code is
  Port ( A : in STD_LOGIC;
         B : in STD_LOGIC;
         Cin : in STD_LOGIC;
         S : out STD_LOGIC;
         Cout : out STD_LOGIC);
end full_adder_vhdl_code;

architecture gate_level of full_adder_vhdl_code is

begin
gate_level:
  S <= A XOR B XOR Cin;
  Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B); 

end gate_level;
Typical HW design flow

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder_vhdl_code is
  Port ( A : in STD_LOGIC;
         B : in STD_LOGIC;
         Cin : in STD_LOGIC;
         S : out STD_LOGIC;
         Cout : out STD_LOGIC);
end full_adder_vhdl_code;

architecture gate_level of full_adder_vhdl_code is
begin
  S <= A XOR B XOR Cin ;
  Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B);
end gate_level;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder_vhdl_code is
  Port ( A : in STD_LOGIC;
          B : in STD_LOGIC;
          Cin : in STD_LOGIC;
          S : out STD_LOGIC;
          Cout : out STD_LOGIC);
end full_adder_vhdl_code;

architecture gate_level of full_adder_vhdl_code is
begin
  S <= A XOR B XOR Cin;
  Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B);
end gate_level;
Typical HW design flow

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder_vhdl_code is
  Port ( A : in STD_LOGIC;
        B : in STD_LOGIC;
        Cin : in STD_LOGIC;
        S : out STD_LOGIC;
        Cout: out STD_LOGIC);
end full_adder_vhdl_code;

architecture gate_level of full_adder_vhdl_code is
begin
  S <= A XOR B XOR Cin;
  Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B);
end gate_level;
```
High-level synthesis goals

Starting from a functional description, automatically generate an RTL architecture

- **Algorithmic description**: no timing notion in the source code
- **Behavioral description**: notion of step / local timing constraints in the source code e.g. `wait()` statements.

Constraints

- **Timing constraints**: latency/throughput
- **Resource constraints**: `#operators`, `#registers`, `#ALM`, etc

Goals

- **Minimization**: resources, latency, consumption, ...
- **Maximization**: throughput
Controller

- FSM controller
- Programmable controller

Datapath components

- Storage components
- Function units
- Connection components

Source: Embedded System Design, (c) 2009, Gajski, Abdi, Gerstlauer, Schirner
This architecture performs the following operations:

- store 2 variables coming from the port P1 in R1 and R2.
- store 1 variable coming from the port P2 in R3
- add the variables stored in R1 and R3, put the result in R4
- add the variables stored in R2 and R3, put the result in R4
High-level synthesis goals

Starting from a functional description, automatically generate an RTL architecture

- Algorithmic description: no timing notion in the source code
- Behavioral description: notion of step / local timing constraints in the source code e.g. wait statements.

Constraints
- Timing constraints: latency/throughput
- Resource constraints: #operators, #registers, #ALM, etc

Goals
- Minimization: resources, latency, consumption, ...
- Maximization: throughput
HLS steps

- **Compilation**
  
  Generates a formal modeling of the specification
HLS steps

- **Compilation**
  Generates a formal modeling of the specification

- **Selection**
  Choose the architecture of the operators
HLS steps

- Compilation
  Generates a formal modeling of the specification

- Selection
  Choose the architecture of the operators

- Allocation
  Define the number of operators for each selected type
HLS steps

- **Compilation**
  Generates a formal modeling of the specification

- **Selection**
  Choose the architecture of the operators

- **Allocation**
  Define the number of operators for each selected type

- **Scheduling**
  Defines the execution date of an operation
HLS steps

- **Compilation**
  Generates a formal modeling of the specification

- **Selection**
  Choose the architecture of the operators

- **Allocation**
  Define the number of operators for each selected type

- **Scheduling**
  Defines the execution date of an operation

- **Binding (or Assignment)**
  Defines which operators will execute a given operation
  Defines which memory element will store a data
HLS steps

- **Compilation**
  Generates a formal modeling of the specification

- **Selection**
  Choose the architecture of the operators

- **Allocation**
  Define the number of operators for each selected type

- **Scheduling**
  Defines the execution date of an operation

- **Binding (or Assignment)**
  Defines which operators will execute a given operation
  Defines which memory element will store a data

- **Architecture generation**
  Writes out the RTL source in the target language e.g. VHDL
Operator Architecture

- **Ripple Carry adder**
  Add two integers $A$ and $B$, cascade of 1-bit adders

- **Carry Select adder**
  Parallelize by speculating carry value
  **Faster but also larger than RCA**
HLS steps

- **Operators Library**
- **Specification**
- **Compilation**
- **Intermediate format**
- **Selection**
- **Allocation**
- **Scheduling**
- **Binding**
- **Architecture generation**
- **RTL architecture**

**Constraints**
- Operators library:
  - Adders: CLA, RCA
  - Multipliers: Booth, Wallace
  - Subtractors: CLA, RCA

**Specification**

\[ O = \left( (n_{01} + n_{02}) \times n_{12} \right) - (n_{21} \times n_{22}) \]

**Intermediate representation**

Credit: Philippe Coussy
HLS steps

Operators Library

Specification

Compilation

Intermediate format

Constraints

Operators library

Adders

multipliers

subtractors

CLA

Booth

Wallace

CLA

RCA

RCA

RCA

Architecture generation

RTL architecture

Specification

\[ O = ((n_{01} + n_{02}) \cdot n_{12}) \cdot (n_{21} \cdot n_{22}) \]

Intermediate representation

Credits: Philippe Coussy
HLS steps

Operators Library

Specification

Compilation

Intermediate format

Selection

Allocation

Scheduling

Binding

Architecture generation

RTL architecture

Constraints

Operators library

Adders

multipliers

subtractors

CLA

Booth

Wallace

CLA

RCA

Specification

\[ O = ((n_{01} + n_{02}) \times n_{12}) \times (n_{21} \times n_{22}) \]

Intermediate representation

Credits: Philippe Coussy
HLS steps

Credits: Philippe Coussy
HLS steps

Credits: Philippe Coussy
Resource constrained HLS

- **Limited number of resources**
  e.g. 2 multipliers, 3 adders

- **Schedule the operators according to available operators in the current control step**

- **Goal**
  Minimize the latency / maximize the throughput

**Allocation and then scheduling**
Time constrained HLS

- **Latency constraint**
  e.g. 5 clock cycles to process all the data

- **Throughput constraint**
  e.g. process each 5 cycles a new set of input data

- **Schedule operations by using operators as much as needed**

- **Goal**
  Minimize the circuit area

**Scheduling and then allocation**
The trouble with loops

\[
\text{for } i := 1 \textbf{ to } n \\
\quad \text{for } j := 1 \textbf{ to } p \\
\quad \quad y[i] += a[i,j] \times x[j]
\]
The trouble with loops

\[
\begin{align*}
& \text{for } i := 1 \text{ to } n \\
& \quad \text{for } j := 1 \text{ to } p \\
& \quad y[i] += a[i,j] \times x[j]
\end{align*}
\]

- No parallelism found!
- **Latency:** $\theta(np)$
- **Throughput:** $\theta(1/np)$
Latency and throughput can be improved with loop transformations

- loop unrolling, loop pipelining
- loop permutation, loop fission/loop fusion, loop skewing, loop tiling, etc
- and composition thereof!

Loop scheduling can be inferred automatically

→ Polyhedral Model
Loop unrolling

\[
\text{for } i := 1 \text{ to } N \\
r[i] = a[i] + b[i];
\]

unroll

\[
\text{for } i := 1 \text{ to } N \text{ step } 4 \\
r[i] = a[i] + b[i]; \\
r[i+1] = a[i+1] + b[i+1]; \\
r[i+2] = a[i+2] + b[i+2]; \\
r[i+3] = a[i+3] + b[i+3];
\]

// + remainder

- Expose loop parallelism
- Reduce latency
- Bounded by the dependence analysis of the HLS compiler!
for $i := 1$ to $N$

$$y[i] = 5 \times x[i] + 3;$$

- Execute the iterations in a **pipelined fashion**
- Block stages are executed in **parallel**
- Improve the **latency** "and the throughput"
for $i := 1$ to $N$
for $j := 1$ to $N$
a[$i,j$] := a[$i-1,j$] + a[$i,j-1$];
Loop skewing

\[
\begin{align*}
\text{for } & \; i := 1 \text{ to } N \\
\text{for } & \; j := 1 \text{ to } N \\
\quad & \; a[i,j] := a[i-1,j] + a[i,j-1];
\end{align*}
\]

\[
\begin{align*}
\text{for } & \; t_1 := 2 \text{ to } 2N \\
\text{for } & \; t_2 := \min\{1, t_1 - N\} \text{ to } \max\{t_1 - 1, N\} \\
\quad & \; (i,j) := U^{-1}(t_1, t_2); \\
\quad & \; a[i,j] := a[i-1,j] + a[i,j-1];
\end{align*}
\]

\[
\begin{pmatrix}
  t_1 \\
  t_2
\end{pmatrix}
= \begin{pmatrix}
  U(
  \begin{pmatrix}
  i \\
  j
  \end{pmatrix}
  )
\end{pmatrix}
\text{ with } U \text{ unimodular e.g. } U = \begin{pmatrix}
  1 & 1 \\
  0 & 1
\end{pmatrix}
\]
Loop skewing

\[
\begin{align*}
&\text{for } i := 1 \text{ to } N \\
&\quad \text{for } j := 1 \text{ to } N \\
&\quad \quad a[i,j] := a[i-1,j] + a[i,j-1];
\end{align*}
\]

\[
\begin{align*}
&\text{for } t_1 := 2 \text{ to } 2N \\
&\quad \text{for } t_2 := \min\{1, t_1 - N\} \text{ to } \max\{t_1 - 1, N\} \\
&\quad \quad (i,j) := U^{-1}(t_1, t_2); \\
&\quad \quad a[i,j] := a[i-1,j] + a[i,j-1];
\end{align*}
\]

- \[
\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = U \begin{pmatrix} i \\ j \end{pmatrix} \text{ with } U \text{ unimodular} \text{ e.g. } U = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}
\]
- Correctness: data dependences must be satisfied
Loop skewing

\[
\begin{align*}
\text{for } i & := 1 \text{ to } N \\
& \text{for } j := 1 \text{ to } N \\
a[i,j] & := a[i-1,j] + a[i,j-1];
\end{align*}
\]

\[
\begin{align*}
\text{for } t_1 & := 2 \text{ to } 2N \\
& \text{for } t_2 := \min\{1, t_1 - N\} \text{ to } \max\{t_1 - 1, N\} \\
(i,j) & := U^{-1}(t_1, t_2); \\
a[i,j] & := a[i-1,j] + a[i,j-1];
\end{align*}
\]

- \[
\begin{pmatrix}
 t_1 \\
t_2
\end{pmatrix}
= U
\begin{pmatrix}
 i \\
j
\end{pmatrix}
\]
  with \( U \) unimodular e.g. \( U = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \)

- **Correctness:** data dependences must be satisfied

- **Generalization:** linear schedules \( \theta(i,j) = A \begin{pmatrix} i \\
j \end{pmatrix} \).
for $i := 1$ to $N$
    for $j := 1$ to $N$
        $c[j] := a[i] + b[j]$;
    for $j := 1$ to $N$
        $d[i] := d[i] + c[j]$;

Loop fusion is possible since data dependences ($c[j]$) are satisfied. After loop fusion, we can eliminate the dimension $i$ of $c$. → less storage, less energy consumption. The inverse transformation allows to parallelize! → trade-off parallelism / storage.
Loop fusion / loop fission

\[
\begin{align*}
\text{for } i &:= 1 \text{ to } N \\
\text{for } j &:= 1 \text{ to } N \\
\quad &\quad c[j] := a[i] + b[j]; \\
\text{for } j &:= 1 \text{ to } N \\
\quad &\quad d[i] := d[i] + c[j];
\end{align*}
\]

\[
\begin{align*}
\text{for } i &:= 1 \text{ to } N \\
\text{for } j &:= 1 \text{ to } N \\
\quad &\quad c[j] := a[i] + b[j]; \\
\text{for } j &:= 1 \text{ to } N \\
\quad &\quad d[i] := d[i] + c[j];
\end{align*}
\]

- Loop fusion is possible since data dependences (c[j]) are satisfied.

→ less storage, less energy consumption

The inverse transformation allows to parallelize!

→ trade-off parallelism / storage
Loop fusion is possible since data dependences \( (c[j]) \) are satisfied.

After loop fusion, we can eliminate the dimension \( i \) of \( c \).

\[ \rightarrow \text{less storage, less energy consumption} \]
Loop fusion is possible since data dependences (c[j]) are satisfied.

After loop fusion, we can eliminate the dimension $i$ of $c$.

$\rightarrow$ less storage, less energy consumption

The inverse transformation allows to parallelize!

$\rightarrow$ trade-off parallelism / storage
for $i := 1$ to $n$
    for $j := 1$ to $p$
        $y[i] += a[i,j] \times x[j]$

Which transformation(s)?
for $i := 1$ to $n$
  for $j := 1$ to $p$
    $y[i] += a[i,j] \times j$
\[\text{unroll}\]

for $i := 1$ to $n$ step 2
  for $j := 1$ to $p$
    $y[i] += a[i,j] \times j$
  for $j := 1$ to $p$
    $y[i+1] += a[i+1,j] \times j$

Which transformation(s)?
A solution

Pipelined parallelism ("dataflow") → process network

- Latency: $\theta(B_1 p)$
- Throughput: $\theta(1/B_1 p)$
A solution with streamed I/O

Step-by-step communications \[\rightarrow\text{systolic process network}\]

- **Latency:** \(\theta(B_1 p + n/B_1)\)
- **Throughput:** \(\theta(1/B_1 B_2)\)

\[\rightarrow\text{I/O interface?}\]
Lessons

Classical HLS flow can be seen as gcc, it produces a reasonable circuit.

Program transformations are required to improve throughput and latency:
  - Exposing parallelism...
  - ... while being aware of I/O constraints

→ automatic parallelization
1. Data transfers: architectures, models and optimizations

2. Front-end: automatic parallelization
   → deriving process networks

3. Back-end: control and channel synthesis
   → mapping process networks to circuits