Lecture 1: Introduction to High-Level Synthesis

CR11 – Hardware Compilation and Simulation
ENS-Lyon – M2iF

Christophe Alias

Outline

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

Typical HW design flow

- Algorithm
- High-Level Synthesis
- RTL
- Logic Synthesis
- Gate Level Netlist
- Layout
- FPGA
- GDS II
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

Example

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
- **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 Assignement)**
  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
  ![Ripple Carry adder diagram](image)
  Specification: \( O = (A \times 2 + B) \mod 2 \mod 2 + C_i \)

- **Carry Select adder**
  Parallelize by speculating carry value
  **Faster but also larger than RCA**
  ![Carry Select adder diagram](image)

---

Credits: Philippe Coussy

Sources: Wikimedia Commons & MIT
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

Credits: Philippe Coussy
The trouble with loops

```
for i := 1 to n
  for j := 1 to p
    y[i] += a[i,j] * x[j]
```

▶ No parallelism found!
▶ Latency: $\Theta(np)$
▶ Throughput: $\Theta(1/np)$

Solution: loop transformations

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

```
for i := 1 to N 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];
```

▶ Expose loop parallelism
▶ Reduce latency
▶ Bounded by the dependence analysis of the HLS compiler!

Loop pipelining

```
for i := 1 to N
  y[i] = 5*x[i] + 3;
```

▶ Execute the iterations in a pipelined fashion
▶ Block stages are executed in parallel
▶ Improve the latency “and the throughput”
Loop skewing

for \( i := 1 \) to \( N \)
for \( j := 1 \) to \( N \)
\[ a[i,j] := a[i-1,j] + a[i,j-1]; \]

\[ \begin{align*}
\text{skewing} \\
\begin{array}{c}
1 \\
N \\
\end{array} & \begin{array}{c}
1 \\
N \\
\end{array} & \begin{array}{c}
1 \\
2N \\
\end{array} & \begin{array}{c}
1 \\
2N \\
\end{array} \\
\end{align*} \]

\[ \begin{array}{c}
\text{skewing} \\
\begin{array}{c}
1 \\
N \\
\end{array} & \begin{array}{c}
1 \\
N \\
\end{array} & \begin{array}{c}
1 \\
2N \\
\end{array} & \begin{array}{c}
1 \\
2N \\
\end{array} \\
\end{align*} \]

\[ \begin{cases}
\{1, N - 1\} & \text{if } t_1 = 2N \\
\{t_1 - 1, N\} & \text{if } t_1 = 2N - 1
\end{cases} \]

\[ (i, j) := U^{-1}(t_1, t_2); \]

\[ a[i,j] := a[i,j] + a[i,j]; \]

\[ \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = U \begin{pmatrix} i \\ j \end{pmatrix} \]

\[ U \text{ unimodular e.g. } U = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \]

\[ \text{Correctness: data dependences must be satisfied} \]

\[ \text{Generalization: linear schedules } \theta(i, j) = A \begin{pmatrix} i \\ j \end{pmatrix}. \]

Loop fusion / loop fission

for \( i := 1 \) to \( N \)
for \( j := 1 \) to \( N \)
\[ a[i,j] := a[i-1,j] + a[i,j-1]; \]

\[ \begin{cases}
\{1, N - 1\} & \text{if } t_1 = 2N \\
\{t_1 - 1, N\} & \text{if } t_1 = 2N - 1
\end{cases} \]

\[ (i, j) := U^{-1}(t_1, t_2); \]

\[ a[i,j] := a[i,j] + a[i,j]; \]

\[ \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = U \begin{pmatrix} i \\ j \end{pmatrix} \]

\[ U \text{ unimodular e.g. } U = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \]

\[ \text{Correctness: data dependences (c[j]) are satisfied} \]

\[ \text{After loop fusion, we can eliminate the dimension } i \text{ of c.} \]

\[ \Rightarrow \text{less storage, less energy consumption} \]

\[ \Rightarrow \text{The inverse transformation allows to parallelize!} \]

\[ \Rightarrow \text{trade-off parallelism / storage} \]

Quizz

for \( i := 1 \) to \( n \)
for \( j := 1 \) to \( p \)
\[ y[i] += a[i,j] * x[j]; \]

\[ \begin{cases}
\{1, N - 1\} & \text{if } t_1 = 2N \\
\{t_1 - 1, N\} & \text{if } t_1 = 2N - 1
\end{cases} \]

\[ (i, j) := U^{-1}(t_1, t_2); \]

\[ a[i,j] := a[i,j] + a[i,j]; \]

\[ \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = U \begin{pmatrix} i \\ j \end{pmatrix} \]

\[ U \text{ unimodular e.g. } U = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \]

\[ \text{Correctness: data dependences must be satisfied} \]

\[ \text{Generalization: linear schedules } \theta(i, j) = A \begin{pmatrix} i \\ j \end{pmatrix}. \]

A solution

\[ \text{Pipelined parallelism ("dataflow") } \]

\[ \Rightarrow \text{process network} \]

\[ \text{Latency: } \theta(B_1 p) \]

\[ \text{Throughput: } \theta(1 / B_1 p) \]

Which transformation(s)?
A solution with streamed I/O

▶ Step-by-step communications → systolic process network
▶ Latency: $\theta(B_1 p + n/B_1)$
▶ Throughput: $\theta(1/B_1 B_2)$
→ 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

Outline

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