Compilation

• Reconfigurable configurable has the ability to execute multiple operations in parallel through spatial distribution of the computing resources

• When compiling a SW-based sequential language like (C) into a concurrent language like Verilog, it is necessary to either
  – Manually instruct the compiler to incorporate parallelism either through special instructions or compiler directives
  – Automatically through the compiler or manually by hand expose the concurrency in the application
Data-flow graphs (DFG)

- A data-flow graph (DFG) is a graph which represents a data dependencies between a number of operations.
- Dependencies arise from various reasons:
  - An input to an operation can be the output of another operation
  - Serialization constraints, e.g., loading data on a bus and then raising a flag
  - Sharing of resources
- A dataflow graph represents operations and data dependencies:
  - Vertex set is one-to-one mapping with tasks
  - A directed edge is in correspondence with the transfer of data from an operation to another one
Consider the following example

[Giovanni’94] Design a circuit to numerically solve the following differential equation in the interval \([0, a]\) with step-size \(dx\)

\[ y'' + 3xy' + 3y = 0 \]
\[ x(0) = x; y(0) = y; y'(0) = u \]

```
read (x, y, u, dx, a);
do {
    x1 = x + dx;
    u1 = u - (3*x*u*dx) - (3*y*dx);
    y1 = y + u*dx;
    c = x1 < a;
    x = x1; u = u; y = y1;
} while (c);
write(y);
```
Data-flow graph example

\[
\begin{align*}
\text{xl} &= x + dx;  \\
\text{ul} &= u - (3*x*u*dx) - (3*y*dx);  \\
\text{yl} &= y + u*dx;  \\
c &= \text{xl} < a;
\end{align*}
\]
Detecting concurrency from DFGs

Extended DFG where vertices can represent links to link graph
DFGs in a hierarchy of graphs

Paths in the graph represent concurrent streams of operations
Control / data-flow graphs (CDFG)

- Control-flow information (branching and iteration) can be also represented graphically
- Data-flow graphs can be extended by introducing branching vertices that represent operations that evaluate conditional clauses
- Iteration can be modeled as a branch based on the iteration exit condition
- Vertices can also represent model calls
CDFG example

\[
\begin{align*}
  x &= a \times b; \\
  y &= x \times c; \\
  z &= a + b; \\
  \text{if } (z \geq 0) \{ \\
    p &= m + n; \\
    q &= m \times n; \\
  \} 
\end{align*}
\]
Behavioral code optimizations

- *Tree-height reduction* applies to arithmetic expression trees and strives to achieve the expression split into two-operand expressions to exploit parallelism.
- The idea is to attempt to balance the expression tree as much as possible.
- If we have $n$ operations, what is the best height that can be achieved?
- Example: $x = a + b \times c + d$
Tree-height reduction

$$x = a \cdot (b \cdot c \cdot d + e)$$

Exploiting the distributive property at the expense of adding an operation
Constant and variable propagation

- *Constant propagation* consists of detecting constants operands and pre-computing the value of the operation with that operand. The result might a constant which can be propagated to other operations as input.

  - **Example:**
    - `a = 0; b = a + 1; c = 2 * b`
    - Replaced by → `a = 0; b = 1; c = 2`

- *Variable propagation* consists of detecting the copies of the variable and using the right-hand side in the following references in place of the left-hand side.

  - **Example:**
    - `a = x; b = a + 1; c = 2 * a`
    - Replaced by → `a = x; b = x + 1; c = 2 * x`
CSE and DCA

• *Common Sub-expression Elimination (CSE)* avoids unnecessary computations.

• Example:
  
  \[- a = x + y; b = a + 1; c = x + y\]
  
  Can be replaced by \[→ a = x + y; b = a + 1; c = a\]

• *Dead code elimination (DCA).* Dead code consists of all operations that cannot be reached, or whose results is never referenced elsewhere. Example:

  \[a = x; b = x + 1; c = 2 * x;\]
  
  The first assignment can be removed if it is never subsequently referenced
Operator strength reduction & code motion

- **Operator strength reduction** means reducing the cost of implementing an operator by using a “weaker” one (that uses less hardware / simpler and faster)

- **Example:**
  - \( a = x^2; \ b = 3 \times x \)
  Replaced by \( a = x \times x; \ t = x \ll 1; \ b = x + t \)

- **Code motion** often applies to loop invariants, i.e., quantities that are computed inside an iterative construct but whose values do not change from iteration to iteration.

- **Example:**
  - for \((i = 1; \ i < a \times b) \{\ldots\}\)
  Replaced by \( t = a \times b; \ \text{for} \ (i = 1; \ i \leq t) \{\ldots\}\)
Control-flow-based transformations

- Control-flow transformations are typically utilized to create more opportunities for data-flow transformations to be exercised.
- *Model expansion* consists in flattening locally the model call hierarchy. Therefore the called model disappears, being swallowed by the calling one.
- A possible benefit is that the scope of application of some optimization techniques is enlarged yielding potentially a better circuit.
- **Example:**
  - $x = a + b; ~ y = a*b; ~ z = \text{func}(x, y)$
  - where $\text{func}(p, q) = \{t = q-p+p*q; \ \text{return} \ t;\}$
  - → By expanding func, we get
    - $x = a + b; ~ y = a*b; ~ z = a - b + a*b$;
  - → CSE $x = a+b; ~ y = a*b; ~ z = a-b+y$;
Conditional expansion

- A conditional construct can be always transformed into a parallel construct with a test in the end.
- Conditional expansion can increase the performance of the circuit when the conditional clause depends on some late-arriving signal.
- However, it can preclude the possibility of hardware sharing.

If (C) then x=A else x=B
\[ \Rightarrow \text{compute A and B in parallel, } x = C \ ?A:B \]
Loop unrolling (expansion)

- In loop expansion, or unrolling, a loop is replaced by as many instances of the body as the number of operations. The benefit is in expanding the scope of other transformations.
- Example:

```c
x = 0;
for (i = 1; i <= 12; i++) {
    x = x + a[i];
}
```

```c
x = 0;
for (i = 1; i <= 12; i = i+3) {
    x = x + a[i] + a[i+1] + a[i+2];
}
```