HIGH-LEVEL SYNTHESIS FOR RECONFIGURABLE DEVICES

High-level synthesis (HLS) deals with the specification of a given application at a very high level of abstraction as well as its implementation on a given platform. The starting point is to specify the application in a given high-level language or tool. For this modelling step, several possibilities exist for capturing the behaviour of a systems, from the very simple Finite State Machines (FSM) and their extensions like State charts, Control Dataflow Graphs, to very complex tools like the Petri Nets, each of which has a different level of powerfulness. In this regard here we will study data flow graph and its two extensions: sequencing graph and finite state machine with data path.

Dataflow Graphs:

A *Dataflow Graph (DFG)* provides a means to describe a computing task in a streaming mode. Given a code segment in a high-level language like C or C++, each operator represents a node in the dataflow graph. The inputs of the nodes are the operand on which the corresponding operator is applied. The output of a node represents the result of the operation on that node. The output of a node can be used as input to other nodes, thus defining a *data dependency* in the graph. Nodes that they depend on others are called *successors* of the nodes on which they depend. Nodes on which other nodes depend are the *predecessors* of the nodes that depend on. Some nodes in the graph have no predecessors and others have no successors. Dataflow graphs might be normalized by inserting in the graph two fake nodes, whose operation has no effect on the dataflow computation. The first fake node is connected as predecessor of all nodes without any predecessor and the second fake node is connected as successor of all nodes that have no successors.

Example: As example of dataflow graph, consider the computation of the quadratic root using the formula ¿ $\frac{(\sqrt{b^2-4ac})-b}{2a}$. The corresponding dataflow graph is shown in Figure 1.

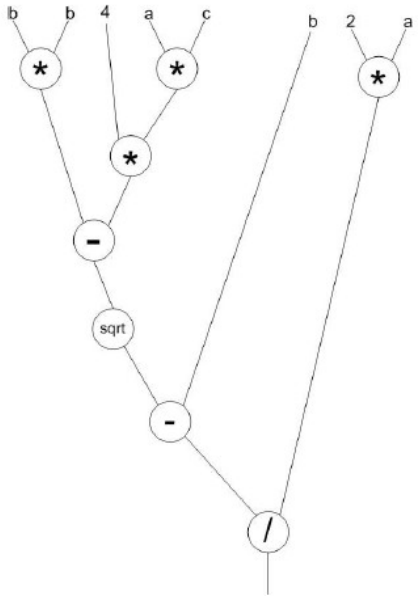

Fig 1: Computational Graph for Quadratic Root

Definition of Dataflow Graph: Given a set of tasks T = {T_1, ..., T_k}

- A dataflow graph (DFG) is a directed acyclic graph G = (V, E), where V = T is the set of nodes representing operators and E is the set of edges.

- An edge e = (vi, vj) ∈ E is defined through the (data) dependency between task Ti and task Tj.

We assume that for each task, an equivalent hardware implementation exists which occupies a rectangular area on the chip. Therefore, the nodes as well as the edges in a DFG possess some characteristics like height, length, area, latency and width, that are derived from the hardware resources used later to implement those tasks. Those values are formally defined as follows:

Given a node $v_i \in V$ and its implementation H_{v_i} as rectangular shape module in hardware.

1. l_i denotes the length and h_i the height of H_{v_i}
2. $a_i = l_i \times h_i$ denotes the area H_{v_i}
3. The latency t_i of v_i is the time it takes to compute the function of v_i using the module H_{v_i}
4. For a given edge e_{ij}= (vi, vj), which defines a data dependency between vi and vj , we define the weight w_{ij} of e_{ij} as the width of bus connecting two components H_{v_i} and H_{v_j}.
5. The latency t_{ij} of e_{ij} is the time needed to transmit data from H_{v_i} to H_{v_j}.

Here v_i denotes a node of the graph and its hardware implementation will be represented by H_{v_i}.

Any program written in a high-level language can be compiled into a dataflow graph, provide that the program is free of loops and branching instructions. This restriction does not match with the reality, since loops and branch instructions are available in the most of the programs, for the evaluation of branching conditions at run-time, and to decide on the segment to be executed according to the value of the condition variables. In case of non-nested loops, the body of a loop is always a set of instructions that can be represented using a dataflow data structure. Several extension of dataflow graph exists to capture program with control structures and loop. We will consider two of them here: the sequencing graphs and the finite state machines with datapath.

Sequencing Graph:

A *sequencing graph* is a hierarchical dataflow graph with two different types of nodes: The *operation nodes* corresponding to normal "task nodes" in a dataflow graph and the *link nodes* or branching nodes that point to another sequencing graph in a lower level of the hierarchy. Linking nodes evaluate conditional clauses. They are placed at the tail of alternative paths corresponding to possible branches. Loops can be modelled by using a branching as a tail of two paths, one for the exit from the loop and the other for the return to the body of the loop, which is the sub-sequencing graph associated with the link node.

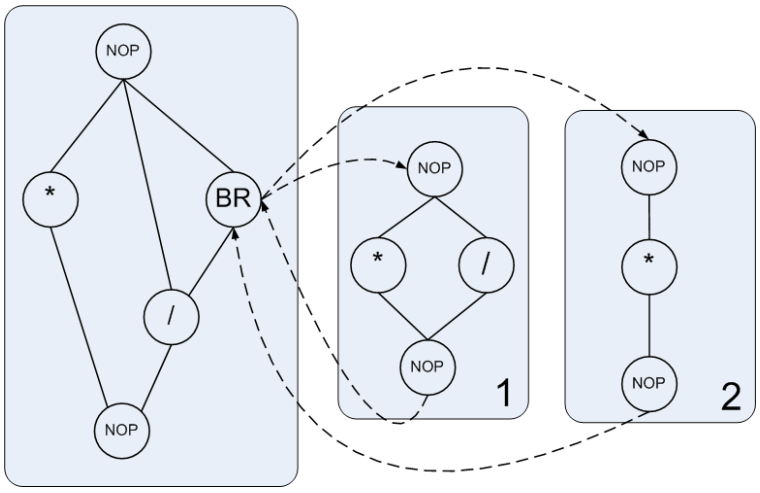

Fig 2: Sequencing graph with a branching node linking to two different sub graphs

Figure 2 shows an example of sequencing graph with a branching node BR containing two branching paths. According to the conditions that node BR evaluates, one of the two sub-sequencing (1 or 2)

graphs can be activated. In order to implement a loop, only one sub-sequencing graph in which the body of the loop is implemented, will be considered. The node BR will then evaluate the exit condition and branch to the next node of the hierarchy level, if the condition holds. Otherwise the body of the loop is re-entered by reactivating the corresponding sub-sequencing graph.

Finite State Machine with Datapath:

Another extension can be done on a dataflow by integrating a finite state machine in the model, in order to control the execution on a datapath defined by the dataflow graph. The result is the so called *Finite State Machine with Data-Path (FSMD)*. A finite finite state machine with datapath (FSMD) can be formally defined as a 7-tuple < S, I,O, V, F,H, s_0 > where:

- S = {s_0, $s_1 \cdots$, s_l} is a set of states.
- I = $\{i_0, i_1, \ldots, i_m\}$ is a set of inputs.
- O = $\{o_0, o_1, \ldots, o_n\}$ is a set of outputs.
- V = $\{v_0, v_1, \ldots, v_n\}$ is a set of variables.
- F: S × I × V → S is a transition function that maps a tupel (states, input variable, output variable) to a state.
- H : S → O + V is an action function that maps the current state to output and variable.
- s_0 is an initial state.

FSMD have some fundamental differences with traditional finite state machines.

a) The transition function operates on arbitrary complex data type like in high-level programming language.
b) The transition and action functions may include arithmetic operations rather than just Boolean operations. The arithmetic operations and the complex data types implicitly define a datapath structure in the specification.

The transformation of a program into a FSMD is done by transforming the statements of the program into FSMD states. The statements are first classified in three categories:

1. ***Assignment statements*:** For an assignment statement, a single state is created that executes the assignment action. An arc connecting the so created state with the state corresponding to the next program statement is created.
2. ***Branch statements:*** For a branch statement, a condition state *C* and a join state *J* both with no action are created. An arc is added from the conditional state to the first statement of the branch. This branch is labelled with the first branch condition. A second arc, labelled with the complement of the first condition ANDed with the second branch condition is added from the conditional state to the first statement of the branch. This process is repeated until the last branch condition. Each state corresponding to the last statement in a branch is then connected to the join state. The join state is finally connected to the state corresponding to the first statement after the branch.
3. ***Loop statements:*** For a loop statement, a condition state C and a join state J, both with no action are created. An arc, labelled with the loop condition and connecting the conditional state C with the state corresponding to the first statement in the loop body is added to the FSMD. Accordingly another arc is added from C to the state corresponding to the first statement after the loop body. This arc is labelled with the complement of the loop condition. Finally an edge is added from the state corresponding to the last statement in the loop to the join state and another edge is added from the join state back to the conditional state.

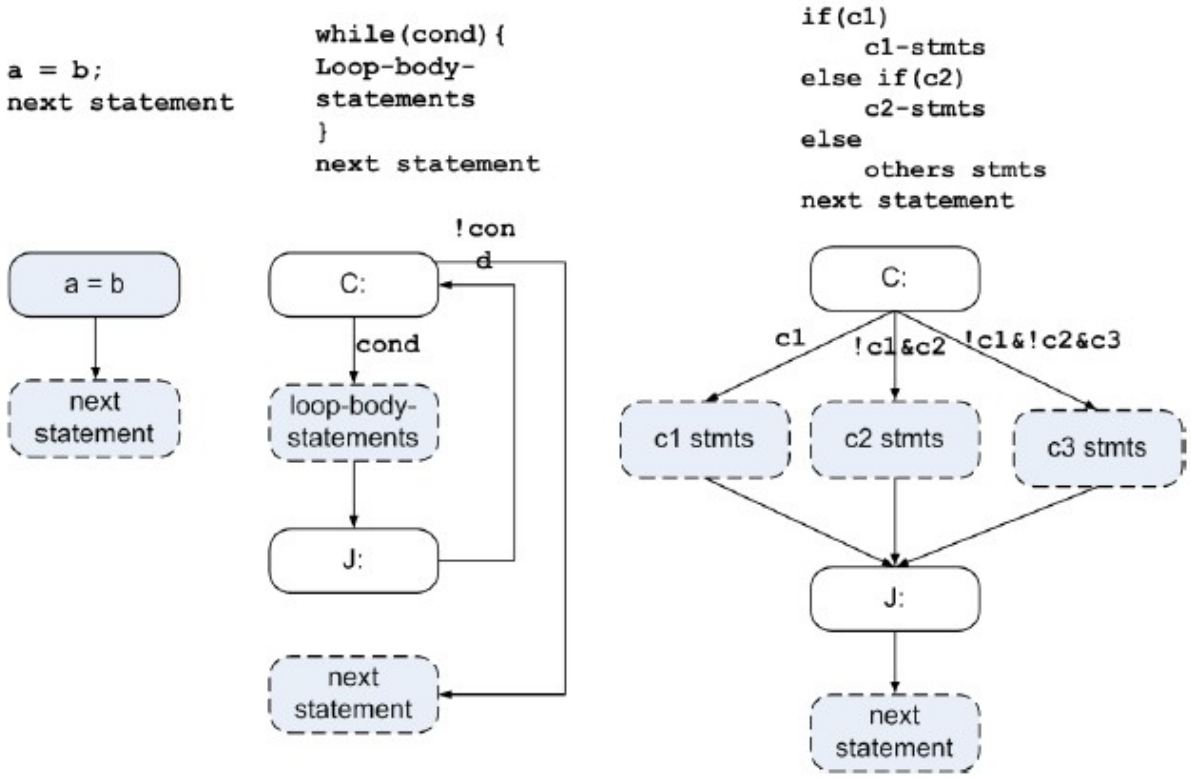

Fig 3: Transformation of a sequential program into a FSMD

Example: Let us model the greatest common divisor GCD of two numbers, using an FSMD . The sequential version of the GCD is given in the algorithm and the corresponding FSMD is shown in Figure 5.

```
1:  variable a, b, gcd: integer;
2:  done := FALSE;
3:  while (!done) do
4:     if (a > b) then
5:        a := a - b;
6:     else
7:        if (b > a) then
8:           b := b - a;
9:        else
10:          done = TRUE;
11:       end if
12:    end if
13: end while
14: gcd := a;
```

Figure 4: Algorithm for *GCD*

The loop state (C1) and the branching state within the loop are white and other states are grey. Also we have labelled the states with the action to be taken in those states. An additional label corresponding to the line of the instruction in the program is placed on each state.

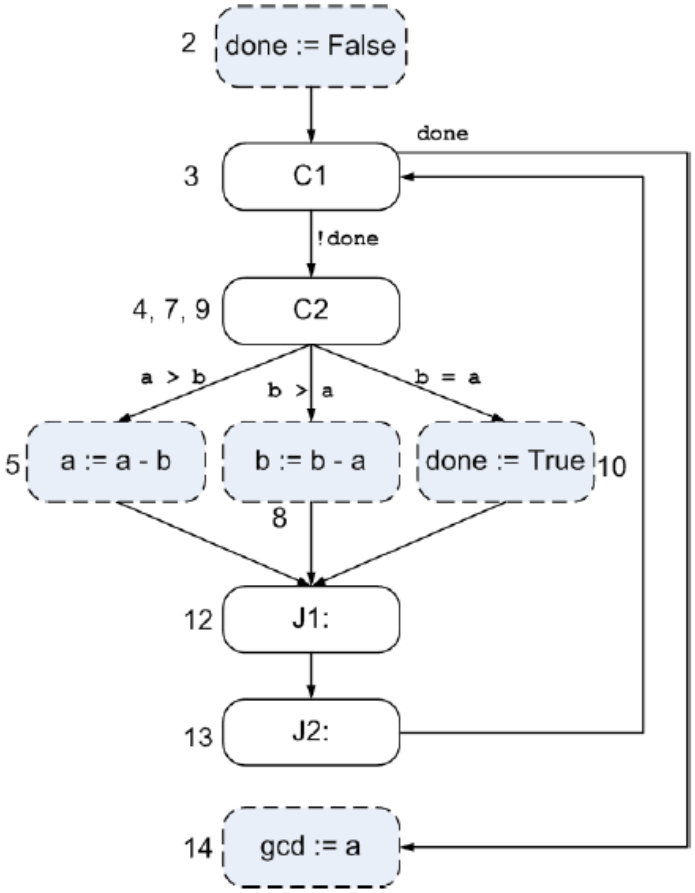

Fig 5: Transformation of the greatest common divisor program into a FSMD

After the program transformation into an FSMD, a datapath must be created that will be controlled by a finite state machine derived from the FSMD. The process of creating the datapath is straightforward.

a) First a register must be instantiated for each variable in the program. An output port implicitly declares a variable for which a register must be created.
b) In the second step, a functional unit will be created for each arithmetic operation in the state diagram.
c) The third step consists of connecting the ports of the functional unit with those of the variable.
d) In the fourth step, a unique identifier is created for each control input and output of the functional units in the datapath. Sharing the operator can be done using multiplexers.

Figure 6 shows the datapath of the greatest common divisor program together with control finite state machine resulting from the FSMD. The control signals are LDA and LDB to load the two registers A and B with the values coming from the subtractor, the SELL and SELR to select the corresponding value from the multiplexer. Those signals are controlled by the FSM that set them according to its current state. The status signals AGTB and ALTB are used by the FSM to decide about the next state to move to, depending on the value of the comparison between A and B.

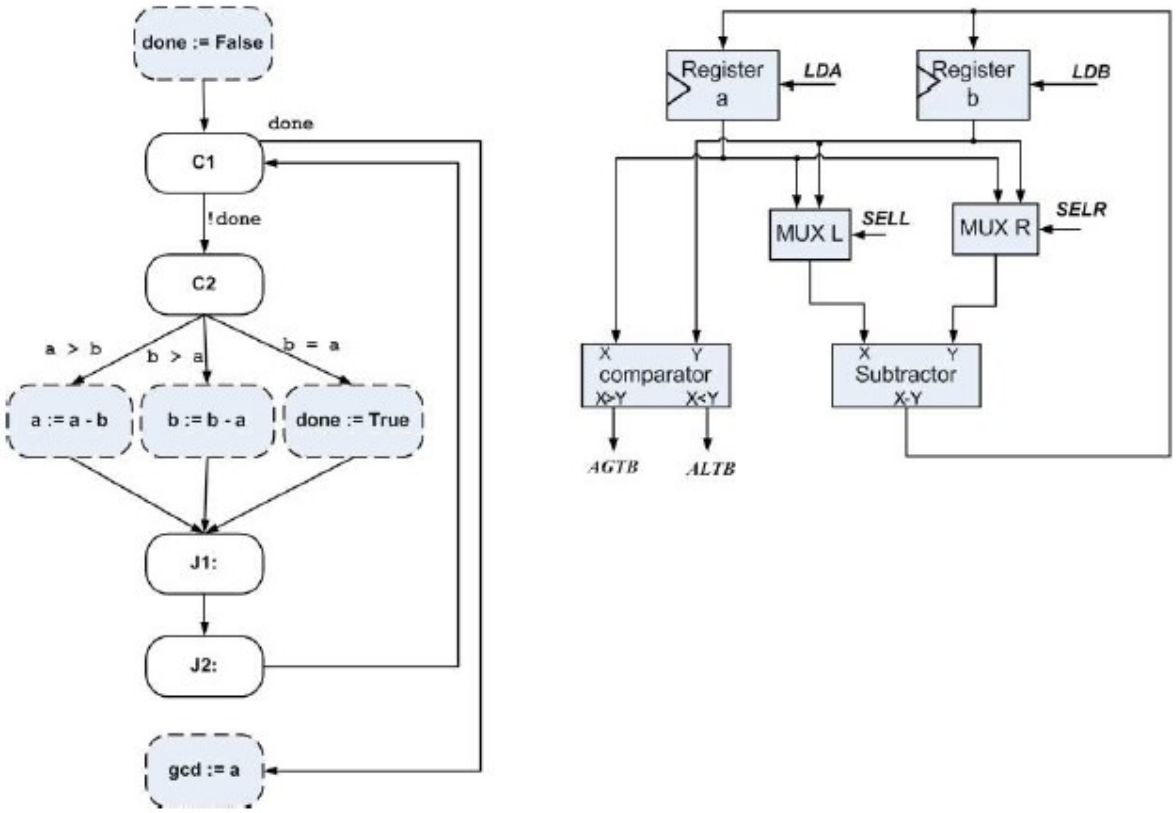

Fig 6: The datapath and the corresponding FSM for the GCD-FSMD

Having specified a system using one of the modelling tools previously presented, the next step will consists on the compilation of the specification into a set of hardware components. This compilation step, also known as synthesis is usually done in three different steps.

The first one is the *allocation*, which defines the type of resources required by the design, and for each type the number of instances. In the GCD case, the types of resources needed are comparators, subtractors, registers and multiplexers. Two registers are instantiated and two multiplexers are instantiated while only one subtractor and one comparator are used.

The next step after the allocation is the *binding*. In this step, each operation is mapped to an instance of a given resource. Since many operators can be mapped to the same instance of a resource, a *schedule* is used in the third step to decide on which operator should be assigned a given resource at a given period of time.

Allocation: For a given specification with a set of operators or task $T = \{t_1, t_2, \cdots, t_n\}$ to be implemented on a set of resource types $R = \{r_1, r_2, \cdots, r_t\}$. An Allocation is a function $\alpha : R \rightarrow +$, where $\alpha(r) = z_i$ denotes the number of available instances of resource type r_i

Binding: For a given specification with a set of operators or task $T = \{t_1, t_2, \cdots, t_n\}$ to be implemented on a set of resource types $R = \{r_1, r_2, \cdots, r_t\}$. A binding is a function $\beta : T \rightarrow R \times +$, where $\beta(t_i) = (r_i, b_i)$, $(1 \leq b_i \leq \alpha(r_i))$ denotes the instance of the resource type r_i on which t_i is mapped to.

Schedule: For a given specification with a set of operators $T = \{t_1, t_2, \cdots, t_n\}$, a schedule is a function $\varsigma : V \rightarrow +$, where $\varsigma(t_i)$ denotes the starting time of the task t_i.