# Bibliographical study: Describing circuits in C

Antoine Martinet

February 5, 2015

# Contents

## Introduction

The idea of designing circuits from high-level programming languages is not new. Historically, engineers used to design circuits using hardware description languages (HDL). Indeed, HDLs are very adapted for circuit design, as they are fully parallel, and extremely low level, so very precise. Thus, HDL are adapted for hardware optimization. Today, the problem faced by designers, as the technology evolves, is the constraints implied by such languages. Indeed, full parallelism can become very complex when the application size grows. Moreover, HDLs are redondant and heavy in terms of code size, so again there are scaling problems when the application grows. Finally, HDLs need a good knowledge of the targetted hardware to be efficently written.

At this point, they are three main purposes of using higher level languages to design electronics. The first one is, design faster, automatizing fastidious processes, but staying aware of hardware concerns. The second one is, broadcast the ability to design hardware to every programmer. The third one is to be able to compute every C code into a physical circuit, that will run much faster than on traditional architectures. The latter case will help the design of application specific integrated circuits (ASICS) and of applications on field programmable gate arrays (FPGA).

All those ideas have economical concerns, such as designing the processors faster, or designing faster applications. Moreover, this has a great interest on the FPGA rising market. Indeed, embedded solutions use more and more FPGAs, because they offer a better tradeof in terms of consumption and price for mean-size markets.

As you will see, the goal of designing circuits in C shall seem very interesting, but there are some dangers that we have to avoid. First, a C programmer, who was not taught logic design, might make some mistakes that lead the program to inefficiency. Second, by abstracting the hardware, we hide all the complexity that leads to a good physical implementation, and it could sometimes be very hard to get the proper design from a C code.

Finally, this question gave birth to a huge research community which goal is to design circuits from relatively high-level languages (in this case the C programming language). Thus, this extremely active domain is High-Level Synthesis (HLS).

In this bibliographical study, I will first try to give the important concerns of designing hardware. Then, I will enlarge the subject to compilation in general, because high-level synthesis raises the same issues. Finally, I will focus on the SSA form that is an important rising feature in today's compilers.

# 1 From HDL to C: the big deal of High-Level Synthesis

## 1.1 HDL and circuit design

Historically, hardware description languages (HDL) were created to automatize the integrated circuits fabrication in fundries. The idea first was to standardize a mean to communicate a circuit specification so that a factory can design every circuit without being rebuilt. The second idea was to be able to simulate circuits behaviour before producing it, so that no error in the final release leads to an unsuable chip. Nowadays, HDL are numerous. The most used ones are of course Verilog (IEEE 1364) and VHDL (IEEE 1076 / IEEE 1164). But we can also cite JHDL (for the java framework), Bluespec (derived from Haskell) and so on [9] [13] [2].

Hence, HDL are not only used for static circuit description, such as ASICs and traditional processors. The use of HDL languages, especially for Verilog and VHDL targets now the rising market of FPGAs. Indeed, FPGAs are fully reconfigurable circuits, so these programming languages are appropriate to such targets.

## 1.2 C programming language

Historically, the C programming language was created to communicate with sequential computing machines. Such computers, like todays classical computers have a static architecture, and a finite instructions set (assembly code) that make a high level application difficult to build. That's why the C programming language has been created, and later the mass of object languages.(ISO/IEC 9899:2011) C comes with memory management primitives, high level data structures (tables, strings, etc...) that doesn't exist at assembly level. Thus they don't exist at the level of architecture. Then it seems very difficult to express the variety of C in an architectural point of view. Indeed, C will here combine both the functional and the architecural point of views. Though, this richness can be achieved adding a huge complexity in the HDL code, because C and HDL are not designed for the same purpose.

## 1.3 High-Level Synthesis

Specifically, a HLS tool compiles the specification of a circuit performing the following actions, that I will describe later, according to P.Coussy, D.DGajski, M.Meredith and A.Takach in [12]:

1. specification compiling
2. ressource allocation (functionnal units, storage componens, buses)
3. scheduling (operation scheduling to clock cycles)
4. operations binding (to functional units)
5. transfers binding (buses)
6. RTL (register transfer level) architecture generation

The scheduling of all these operations is described on figure 1.

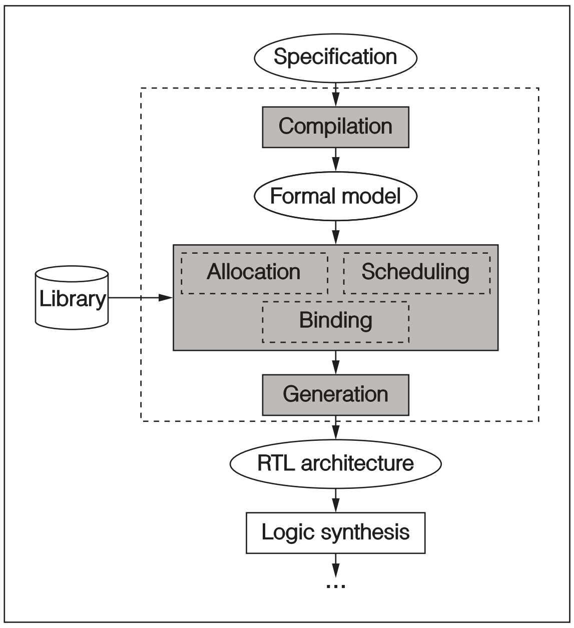


Figure 1: High-Level Synthesis design steps from [12]

### 1.3.1 Ressource allocation

Allocation is the step where the type and number of hardware ressources are determined. In HLS tools, the components are picked in IP-cores libraries, also called RTL component libraries. In the case of compiling on FPGAs, of course, the ressource allocation step has to ensure that used ressources don't exceed available ressources. In the scope of hardware generation (ASICs or other chips), infinite ressources can be considered, but the produced target might become costly. Allocation should also be aware of ressource placement and routing, and should also take care of power consumption/overheating issues.

### 1.3.2 Scheduling

The scheduling step is primordial to have a working hardware. Indeed, there is always a sequentiality in the computation chain induced by the specification. (eg. $a = b + c$ needs $b$, $c$ and then $b + c$ to be available/read/computed before the affectation to $a$ is done). Moreover, each component might not execute in one clock cycle. Nowadays, parallel computing becomes a requirement for computation efficiency. It becomes necessary to take both instruction and block parallelism into account while scheduling the operations. Moreover, scheduling tasks has to guarantee that each signal (ie. component result/output) is provided in time when it is needed.

### 1.3.3 Binding

Variables have to be bound to storage components (registers, memories, disks), and they have to be bound efficiently, so with good locality, using caching potential.

**Sequentiality, design flows** Some of these operations can be done simultaneously, such as Allocation, Scheduling and Binding. There are advantages for doing them at the same time. Indeed both works are binded to each other. For example, tasks that are scheduled just one after the other should remain relatively close together on the final circuit. We can either do them sequentially, to achieve objectives with fixed priority. Indeed, if we do the allocation step first, we can try to reduce the maximum latency (maximum delay between two ends of a communication canal) with the scheduling step. To reduce area under timing constraint, we will try to do the allocation during the scheduling. As said before, ressource-constrained approaches should be used when we are working with limited ressources/budget. On the contrary, time-constrained

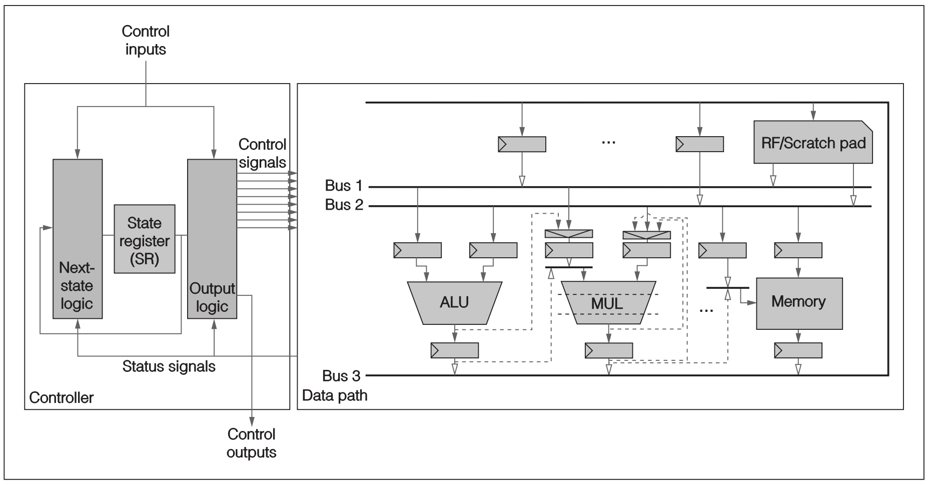


Figure 2: Typical architecture from [12]

approches will try to reduce the circuit area (to reduce the latency) while sticking to time and throughput requirements.

### 1.3.4 RTL generation

The last three steps often use libraries for using proper algorithms to complete efficiently. Then, we can get an operator topology and need now to implement it physically. The RTL architecture is implemented as a set of register-transfer components. The generated architecture often takes the shape of a controler and a data path that consists of a set of functional units and interconnect components. (see Figure 2) Now, some other models are rising, such as dataflow models, that don't follow the classical architecture model. Indeed, we can synthesize a pipelined architecture or any other model. But the principle remains the same. In the case of FPGAs, the RTL model is compiled into a bitstream, that is then pushed on the FPGA so it configures the right way.

## 1.4 Tools for High-Level synthesis

They are many High-Level synthesis tools, some of the most famous were developped by big FPGA's vendors, such as Xilinx (Vivado suites) or Altera (OpenCL SDK). The latest HLS tools usually use C-like input code for the specifications, such as ANSI C, C++, Mentor's Catapult C, Forte's Cynthesizer, Nec's CyberWorkbench, Synfora's PICCO, and Cadence's C-to-

Silicon. High-level synthesis can also be performed by MathWork's Matlab and Simulink. Many use IP approaches to generate the hardware, that can be defined as a dataflow model (black boxes wired together). The alternative to library-based (or IP) approach is to use a configurable processor approach, like in CoWare's Processor Designer and Tensilica's Xtensa. [6] [4] [5]

## 2 Generalities about compilation

The compiler is usually organised as a front-end, an optimizer and a backend. Although I will not describe the backend issues, the front end and the optimiser will be very interesting to look at.

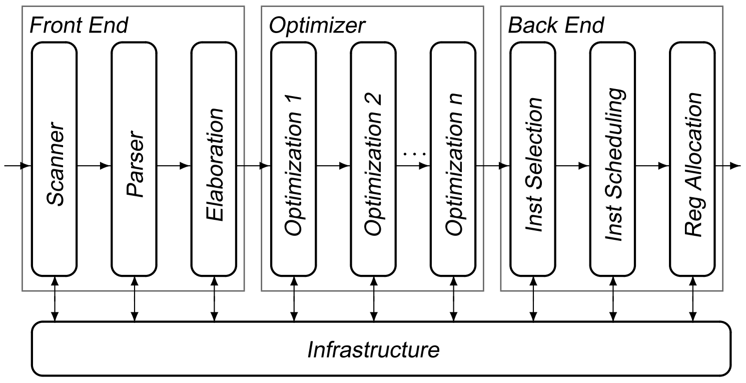


Figure 3: Structure of a typical compiler from [10]

### 2.1 Front ends

The main purpose of the front end is to understand the source program. So it has to know entirely the possibilities of the input language, in order to extract the entire knowledge on the program from the source code. Then, it will be able to encode this knowledge in an intermediate representation (IR) that will be more practical to explore or refactor.

The front end part can be decomposed in three parts, as the optimizing part and the backend part. These parts can be sum up in the previous schema, from [10] (figure 3).

Here, the scanning step is responsible for the syntax checking. Then the parsing step will take care of the semantic. Finally, the elaboration step fills an intermediate representation on which optimizations can be done.

Many front-ends modules have been developped so far. Well known parsers for c or c++ might be lex/flex and bison/yacc, that are not full front-ends as they just perform up to the parsing step. To this they use grammar-based technologies.

### 2.1.1 Parsing : grammars

First, we have to look at formal languages to understand the whole complexity of translating a high-level language into executable code or physical circuit description. Over the world of compilers, two main formal grammars exist: LR- and LL grammars. Formal grammars are defined as sets of production rules, that is, the reading of character produces a set of actions to perform. In a compiler, the language is first read by an automaton following the grammar's rules. Rules of a grammar are processed recursively, so we use derivation to interpret sentences. A rule is composed of two parts: the left-side and the right-side. The leftside describes the starting expression and the second side the decomposition of the expression. At this step, we must define the notion of terminality of symbols. A symbol is terminal if it cannot be derivated. A non-terminal symbol can be derivated in an association of terminal and/or non terminal symbols. Then, we can distinguish two possible derivations, rightmost derivation, when we always derivate the rightmost nonterminal, and the leftmost derivation, when we always derivate the leftmost nonterminal. This is the signification of respectively R and L in LR and LL. For the parsing step, it leads to different algorithms to use for LR or LL grammars. Indeed, implicitely, LR grammars will be parsed by a bottom-up algorithm, that will read the language without any backtracking or guessing. LL parsers, on the contrary, uses lookahead tokens to evalute the expressions. They are of course top-down parsers, which are much easier to construct, but they can just do a subset of the work done by an LR-parser.

### 2.1.2 LLVM front end

One of the most promising front ends today is the clang framework. The purpose of this framework is to provide a front-end for the LLVM compiler. Supported languages are be C, C++, Objective C, and Objective C++. Clang provides optimization techniques, such as static analysis, code generation, refactoring and source to source optimization. LLVM provides also further SSA-based optimizations, which I will describe later. Clang has also an indexing capability (doxygen, jump to definition). It can perform source analysis for bug tracking and fixing. It can also run performance analysis.

But it runs also much faster and less memory-greedy than the well known gcc compiler.

## 2.2 Optimizing

During the parsing step, an intermediate representation is built. Optimizations can then be performed according to the objective(s) of the compiler. We can for example perform optimizations according to the targetted architecture. If we want to compile the language to generate hardware, optimizations can be performed to this goal. We can also create several intermediate representations according to the optimizations we need to do.

### 2.2.1 Intermediates representations

There are several types of intermediate representations (IR) as described in [4] (Chapter 5). To sum up, we can find these categories:

- Graph-based IRs that represent the knowledge in a graph.
- Linear IRs, that is a pseudo-code-like form. Algorithms then iterate linearly on such IRs.
- Hybrid IRs, which combines the two representations, trying to keep advantages and avoid disavantages of both.

About graphical IRs, the most important notion is the notion of control flow graph. It models the flow control between basic blocks in a program. A control flow graph is a directed graph $G = (V, E)$, where each vertex $v \in V$ is a basic computation block and each $e = (v_i, v_j) \in E$ is an eventual flow transfer from $v_i$ to $v_j$ We can then derive a run-time control from graph that will be interesting for just-in-time (JIT) compilers, for example. You can see an example of such a representation on figure 6.

### 2.2.2 Type inference

A compiler may provide many data types, or even permit the programmer to create custom data types. Most of the basic data types are supported directly by the processors, such as integers, floating point numbers, characters and so on. Other data types have to be divided into several smaller parts that are processed differently. (for example complex numbers) This adds some code rewriting during the translation process, and may appear in the IRs hierarchy levels. Though, many languages do not force the programmer to define types for variables. Then the type is most of the time discovered at the first instantiation of the variable. So the compiler has to check this at compilation time. This may cause problems when the compiler does the type checking step: it has to keep all the knowledge on variables in memory. When dealing with functions, the type infrerence problem can take many shapes. First, if we deal with a function that simply has a constant and known type, we can solve the problem using a fixed-point algorithm. If the type remains unknown (when driven by an expression for example), we need to do unification. When the type is changing over the time, then none of these solutions will work. We have then to infer type-changing operations, and infer types when they do change. Renaming affectations operations is there a basis to get proper efficiency.

### 2.2.3 Code shapes

To achieve complex operations, the compiler have to schedule them and allocate them. Indeed, since we are translating in machine code, there are a huge amount of choices to do. For example a simple 3-operand addition has up to 4 possible execution trees, as described in [10]. Indeed, in this example, even if a traditional processor might not be able to perform a 3-operand addition, when we generate hardware, this solution might be the most interesting one in terms of computation time. Here we can see the importance of associativity, that will implicitely decide what execution tree to choose in many cases. This is why the compiler has to be done carefully to prevent interpretation mistakes. Some compilers offer the possibility for the programmer to choose what associativity to look at. This may permit full efficiency in the final computation. But it needs also the programmer to be aware of how to compute the operations efficiently.

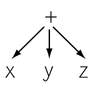
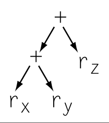
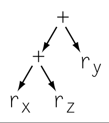
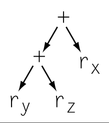
| | Source Code | Low-Level, Three-Address Code | | |
|---|---|---|---|---|
| **Code** | x+y+z | $r_1 \leftarrow r_x + r_y$<br>$r_2 \leftarrow r_1 + r_z$ | $r_1 \leftarrow r_x + r_z$<br>$r_2 \leftarrow r_1 + r_y$ | $r_1 \leftarrow r_y + r_z$<br>$r_2 \leftarrow r_1 + r_x$ |
| **Tree** |  |  |  |  |

Figure 4: Alternate code shapes for x+y+z from [10]

### 2.2.4 Loop unrolling

When trying to get a parallel implementation, the big problem the compiler faces is loops unrolling. Indeed, as non-recursive procedures calls can be done easily in parallel, loops are difficult objects to deal with, because iterations are interdependent and we often do not know how to see the parallelism in them. Extracting parallelism requires analysis of dependencies. We need here to know which atomical computation is needed by other computations. To achieve this, we look at indices to get dependencies of inside-loop computations. Then, we try to extract the parallelism of the loop, that is, computations that can be done without any previous computations. This case is rare in general, so we try to extract the computations that have the less prerequisites and execute them first. Then we aggregate, and we can reduce the latency of the loop. Here is a simple example, taken from [8]

```
for (i = 1; i <= 60; i++) a[i] = a[i] * b + c;
```

```
for  (i = 1; i <= 60; i+=3)
{
   a[i] = a[i] * b + c;
   a[i+1] = a[i+1] * b + c;
   a[i+2] = a[i+2] * b + c;
}
```

Figure 5: Loop unrolling: the second “for” loop is the 2-unrolled version of the first one. Example from [8]

Here, the loop is simple and there is no dependency between iterations, as iterations are only on indices of the table and we do not need any data from previous calculations. Thus, this loop can be unrolled until the end of iterations (that then can all be done in parallel). On traditional architectures, we of course do not have 60 processors, but this might makes sense to unroll completely on an FPGA, for example. More common examples are similar to the following (Figure 5), picked from [8]. Extracting parallelism can then be very hard to achieve. This is however a big research chunk for which the polyhedral model has been created.

```
int c[16] = {};
void test(int* y, int* x, int n) {
       int i = 0;
       for (i = 0; i < 16; i++) {
              y[n] += x[n-i]*c[i];
       }
}
```

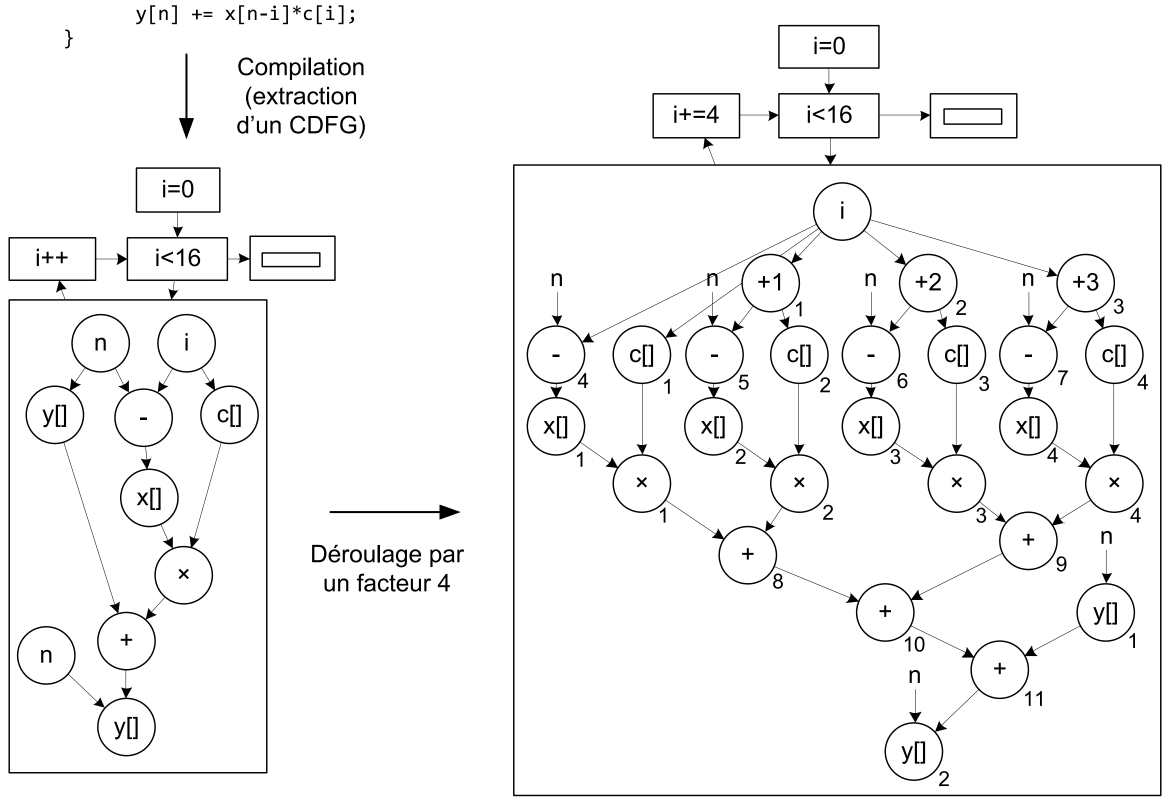


Figure 6: Example of nested loop (upper left), compiled into a CDFG (Control DataFlow Graph) (lower left), unrolled by a 4-factor (lower right). Example from [1]

*Remark* 1. The polyhedral model tries to extract parallelism from an abstracted point of view. The idea is to extract a set of dependencies in a loop [1].

# 3 Compiling just right: focus on the SSA form

Static Single Assignment form (SSA) is a property of an intermediate representation. This is the rising feature in today's compilers, such as gcc and LLVM, and we will see it enhances the power of many optimizations techniques.

SSA requires each variable to be assigned exactly once and before it is used. To obtain such a form from the source code, some operations are needed.

## 3.1 Data-Flow analysis for optimization

To reason about runtime flow values and locate opportunities for optimization and prove the safety of transformations, compilers use dataflow analysis. First, the program is transformed into a control flow graph (CFG). This CFG representation will first allows to do constant propagation. Then many optimizations can be done on the CFG, to get a proper and efficient code.

For instance, we can do live-variable analysis. The idea is to determine from which point the variable has to be alive (i.e. ready to use) and when it's not, so we don't allocate a register for this variable when it is not alive (and we save ressources). This allows to perform mermory optimizations. Moreover, we can try to see exactly when a variable is used to free the memory when it is dead. This allows to compute just right and with even more efficiency. It allows also to eliminate dead code (when a variable is used only once and never released for example) or give feedback errors (when a variable is used but never declared).

At this step, it is important to see that the CFG representation assumes that every path is actually feasable. In the real world, some paths might never been used for certain runs of the program. So optimizations are not always possible, for example dead code elimination might not be really decidable with dataflow analysis.

On the contrary, the compiler must not do too strong inferences, for the same reasons: information from the CFG is often more than runtime information, essentially because the CFG carries absolute information.

One of the today problems in the research community is how to deal with pointers. Indeed, pointer arithmetic is processed as classical arithmetic. The problems here are multiple: not taking pointers into account may force the compiler to declare all the variables alive. An other example is, it prevents the compiler to store pointed values into registers, because it will not be able to prove that a pointer points onto a register or anything else just looking at the assignments: it has to check the value of the assignments.

Another problem is when we deal with procedure calls. Indeed, the compiler has to know how variables are manipulated inside the procedure. And calls within procedure will produce recursive checks that will make the job harder.

Other problems with dataflow analysis are for example expression availability, that is related to global code redundancy elimination. When we have an expression, we want to keep the result if it is reused in the following code, in order to reduce computations. This can be resolved by locally hashing some results and recalling them when needed.

Sometimes the compiler has to know where a definition is done. This could become a little bit complicated when the definition is propagated through several paths in the CFG until it reaches the node processed. Although this operation is quite simple, it requires the mapping of names to definitions, which is relatively hard (costly) to get.

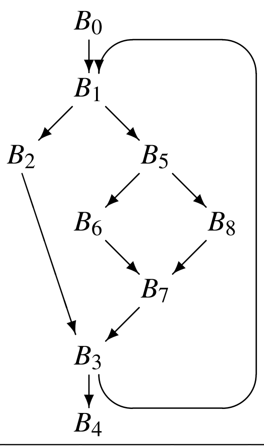
## 3.2 From CFG to SSA Form

The particularity of SSA form is, it encodes many properties from both data flow and control. To get exactly one assignment per variable, SSA includes new definitions/assignments in the code. Then, we can ensure that at each step the value of a variable $x$ has effectively been assigned. The right side of the assignments are $\phi$-functions, that contain the SSA names of variables. These assignements combine the values from distinct edges, and so everyone is unique. An example of a SSA Form program is given figure 8.

### 3.2.1 The Notion of Dominance

At this step, we have to introduce the notion of dominance in the CFG symbolising the program to execute. The dominant set of a node $b_i$ in the control graph is the set of nodes which are on every path from the root to $b_i$ (including $b_i$) We usually denote the dominance $DOM(b_i)$ Here is an example from [10]

**Dominance**
In a flow graph with entry node $b_0$, node $b_i$ *dominates* node $b_j$, written $b_i \underline{\gg} b_j$, if and only if $b_i$ lies on every path from $b_0$ to $b_j$. By definition, $b_i \underline{\gg} b_i$.



| | $B_0$ | $B_1$ | $B_2$ | $B_3$ | $B_4$ | $B_5$ | $B_6$ | $B_7$ | $B_8$ |
|---|---|---|---|---|---|---|---|---|---|
| **DOM(*n*)** | {0} | {0,1} | {0,1,2} | {0,1,3} | {0,1,3,4} | {0,1,5} | {0,1,5,6} | {0,1,5,7} | {0,1,5,8} |

Figure 7: Dominance definition and example from [10]

To achieve the dominance computations we use a fixed-point algorithm that converges reasonably fast. This algorithm is well described in the ninth chapter of [10]

### 3.2.2 Bulding The SSA Form

When the control enters a block, all $\phi$-functions are executed concurently. And they evaluate correspondingly to the edge along which control entered the block.

To achieve this goal, a compiler inserts appropriate $\phi$-functions for each variable, and renames variables to make the two following rules hold:

1. each definition in the procedure creates a unique name
2. each use refers to a single definition.

(definitions from [10])

This two-step procedure results to the following example, figure 9.

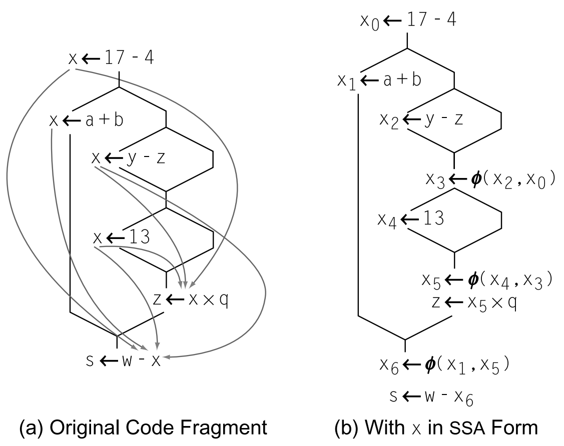


Figure 8: SSA: Encoding Control Flow into Data Flow, from [10]

So to get the SSA form of a program, the compiler first inserts $\phi$-functions, and then processes the renaiming step according to the $\phi$-functions.

The maximal SSA form contains too many $\phi$-function to be efficient. So the compiler has to chose at which block join it is useful to insert them. To be quick, a *phi*-function are inserted only when there is a redefinition inside a single path. $\phi$-functions are also inserted outside the region dominated by a node (of definition).

Dominance frontiers are then computed in 2 steps. First, we compute dominator trees for all nodes. Then, dominance frontiers can be built looking at join points and dominator trees. After this step, $\phi$-functions can be efficiently inserted at each point when control passes through a dominance frontier.

Finally, the renaming part goes from top to bottom, renaming with a counter. A stack is added to deal with multiple counters and variable death.

## 3.3 From SSA Form to executable code

Getting back to executable code, there are a few problems to avoid or work around. The problems come when optimizations and code rearrangement and renaming have been processed. Indeed, dropping subscripts on variable names (passing from the SSA name $a_0$ to the original in-code name $a$ for example) will produce most of the time incorrect code.

To solve this problem, we can insert copies of variables during execution of the $\phi$-functions, but this is not always possible, in the case of a diverging data flow, for example: we will insert undesired copies of the same variable and probably introduce dead code and errors.

Even in working cases, some problems remains here, and for example the Lost-Copy problem. This problem appears when we optimize using an SSA-based algorithm. It can appear that critical edges in the CFG are splitted. For example with loops, when we fold copies and split a critical edge that couldn't be splitted. The following example shows a simple example:

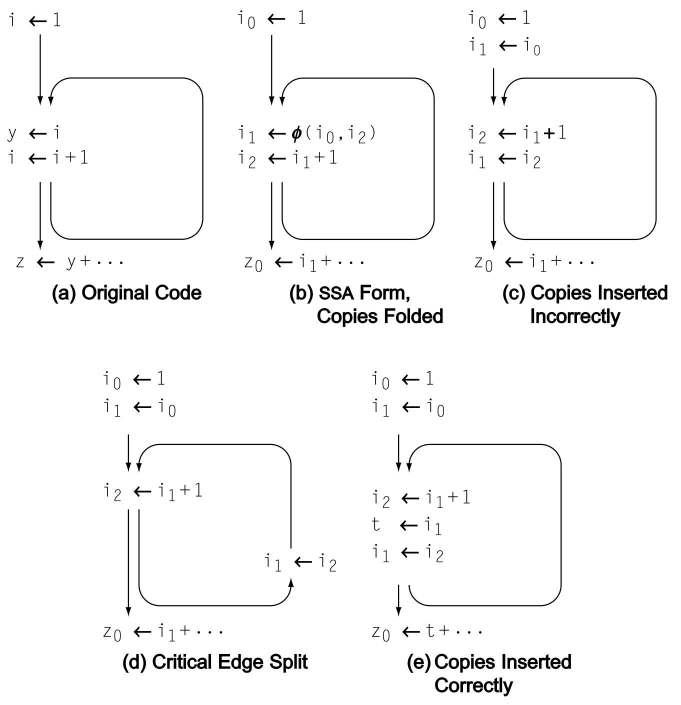


Figure 9: An example of lost-copy problem from [10]

"In panel $b$, the compiler has converted the loop into ssa form and folded the copy from $i$ to $y$ , replacing the sole use of $y$ with a reference to $i_1$ . Panel $c$ shows the code produced by straightforward copy insertion into the $\phi$-function's predecessor blocks. This code assigns the wrong value to $z_0$ . The original code assigns $z_0$ the second to last value of $i$ ; the code in panel $c$ assigns $z_0$ the last value of $i$ . With the critical edge split, as in panel $d$, copy insertion produces the correct behavior. However, it adds a jump to every iteration of the loop." [10]

Here the jump adds latency, so the optimization has to be relativized.

Another problem is the swap problem, that occur when $\phi$-functions are executed concurently. The following problem can happen:

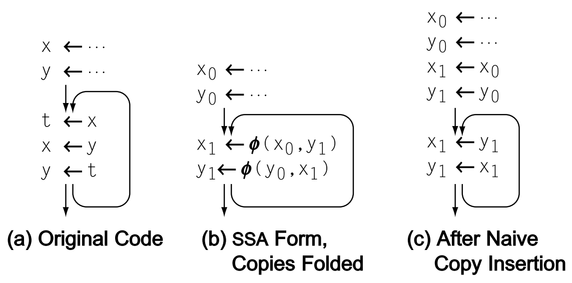


Figure 10: An example of the swap problem from [10]

"Panel a shows the original code, a simple loop that swaps the values of x and y . Panel b shows the code after conversion to ssa form and aggressive copy folding. In this form, with the rules for evaluating $\phi$-functions, the code retains its original meaning. When the loop body executes, the $\phi$-function parameters are read before any of the $\phi$-function targets are defined. On the first iteration, it reads $x_0$ and $y_0$ before defining $x_1$ and $y_1$ . On subsequent iterations, the loop body reads $x_1$ and $y_1$ before redefining them. Panel c shows the same code, after the naive copy-insertion algorithm has run. Because copies execute sequentially, rather than concurrently, both $x_1$ and $y_1$ receive the same value, an incorrect outcome." [10]

Here, splitting the back edge won't help because it leads to the same erroneous copies. The solution here is to follow a two-stage protocol. The first stage copies each argumet on its temporary name, and the second stage copies the values into the targets. But the assignment operations are doubled, and we can't afford such an exponentiation in more complex cases. In the general case, the compiler detects references of $\phi$-functions targets referenced by other $\phi$-functions, and inserts the proper copies to break cycles. A scheduling of copies is then applied to respect $\phi$-functions dependencies.

## 3.4 SSA-based Optimizations

There are many powerfull optimizations doable whith an SSA-based IR. These are often called sparse analysis. First of all the global value numbering (GVN) permits to eliminate redundancies looking for equivalent expressions. This one of course implies dead code elimination, code rearrangement and instruction selection. A good basis about this subject is provided by [11]. This ability is very interesting in the context of high level synthesis because it helps to reduce the size of the final circuit avoiding to duplicate some components instanciations.

Then, we can see easily that the SSA form allows constant propagation analysis. Here again, we are dealing with dead code elimination and instruction selection. This process is well described in [3]

Finally, Liveness analysis is then also possible through SSA-form and more easier as each single value for an assignment is accessible from a unique identifier. A good reference for this seems to be [7].

## Conclusion

In this work, I first tried to explain the outcomes of High-Level synthesis, and its strong binding with compilation. From this point, I came a little deeper into the compilation domain, to finally focus on the SSA form, as it seems to be for now the most promising feature compiler should develop. The interest of taking a look at compilers is, every single optimizations are performed by the compiler. The power of a High-Level synthesis tool is indeed fully due to the power of the compiler, that should be able to process all the knowledge from the high-level source code to produce efficient and constraint-compliant hardware. Back to hardware generation, some problems remain, especially with the use of pointers. Indeed, since most compilers avoid putting pointed values into registers on traditional architectures, the problem becomes much harder when generating hardware. Indeed, we cannot decide where the value of a pointed variable will be located on the generated hardware unless we do specific pointer analysis. Anyway, the problem of defining constraints still remains. Indeed, the decision of what constraint (time, heat, size, and so on) the hardware has to hold, should never be left to the compiler. The compiler remains a tool that processes the code the way the programmer decided to. So the programmer still has to be aware of hardware issues when programming in a high-level language. Moreover, accounting for multiple constraints can still be difficult for a compiler. There is a lot of work to do in High-level synthesis tools, and the compilation step is full part of the job.

## References

[1] A.Morvan. Utilisation du modèle polyhédrique pour la synthèse d'architectures pipelinées. In *Utilisation du modèle polyhédrique pour la*

*synthèse d'architectures pipelinées (Thesis)*, 2013.

[2] B.E.Nelson. The mythical ccm: In search of usable (and reusable) fpga-based general computing machines. In *IEEE 17th International Conference on Application-specific System, Architectures and Processors*, 2006.

[3] C.Click and K.D.Cooper. Combining analyses, combining optimizations. In *ACM Transactions on Programming Languages and Systems, 17(2)*, 1995.

[4] P. Coussy and eds. A. Morawiec. High-level synthesis: From algorithm to digital circuit. In *Springer*, 2008.

[5] J.P. Elliot. Understanding behavioral synthesis: A practical guide to high-level design. In *Kluwer Academic Publishers*, 1999.

[6] D.MacMillen et al. An industrial view of electronic design automation. In *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 2000.

[7] A.Darte F.Brandner, B.Boissinot, B.D.de Dinechin, and F.Rastello. Computing liveness sets for ssa form programs. In *Research report 7503, INRIA*, 2011.

[8] T.Leng J.C.Huang. Generalized loop-unrolling: a method for program speed-up. In *IEEE Symposium on Application-Specific System and Software Engineering Technology (ASSET99)*, 1999.

[9] J.Mermet. Fundamentals and standards in hardware description languages. In *Springer Verlag*, 1993.

[10] L.Thorczon K.D.Cooper. Engineering a compiler. In *MK-editions*, 2012.

[11] L.T.Simpson. Value-driven redundancy elimination. In *Value-Driven Redundancy Elimination (Technical Report), Rice University*, 1996.

[12] M.Meredith P.Coussy, D.DGajski and A.Takach. An introduction to high-level synthesis. In *IEEE Design and Test of Computers*, 2009.

[13] S.P.Jones P.Hudak, J.Hughes and P.Wadler. A history of haskell: being lazy with class. In *The Third ACM SIGPLAN History of Programming Languages Conference, San Diego, California*, 2007.