# Chapter 1

# Hardware Compilation using SSA —(*Pedro C. Diniz and Philip Brisk*)

**Abstract.** This chapter describes the use of SSA-based high-level program representation for the realization of the corresponding computation using hardware circuits. We begin by highlighting the benefits of using a compiler SSA-based intermediate representation in this mapping using an illustrative example. The following sections describe hardware translation schemes for data-paths and the corresponding control logic using SSA intermediate representations. We conclude with a brief survey various hardware compilation efforts both from academia as well as industry that have adopted SSA-based internal representations.

?

layout:
- example
- ???
- survey existing tools

## 1 Brief History and Overview

It looks like in your mind you make a continuum between programmable processors and circuits with between FPGA.

Hardware compilation is the process by which a high-level language, or behavioral, description of a computation is translated into a hardware-based implementation i.e., a circuit expressed in a hardware design language such as VHDL [2] or Verilog [24] which can be directly realized as an electrical (often digital) circuit. ~~Hardware compilation has a long rich history, since the early 70's, if not earlier, first as academic efforts in the context of VLSI design and later as high-level synthesis techniques successfully raised the level of abstraction of the computations the techniques were able to tackle. Reflecting its growing importance as productivity tools many of the high-level techniques have successfully migrated into commercial products for high-level hardware synthesis still used today in industry.~~

Hardwate synthesis
=> VHDL | Verilog

While, initially, developers were forced to design hardware circuits using schematic-capture tools, high-level behavioral synthesis allowed them over the years to leverage the wealth of hardware-mapping and design exploration techniques to realize substantial productivity gains. As an example Figure 1 illustrates these concepts of hardware mapping for the computation expressed as `x = (a * b)` `(c * d) + f`. In figure 1(b) a graphical representation of a circuit that directly implements this computation is presented. Here there is a direct mapping of hardware operator such as adders and multipliers to the operations in the computation. Values are stored in register and the entire computation lasts a single (albeit long) clock cycle. The input data values are stored in the input registers on the left and a clock cycle later the corresponding results of the computation are ready and saved in the output registers on the right-hand-side of the figure. Overall this direct implementation uses two multipliers, two adders/subtractors and six registers with a very simple control scheme store s the input in the input registers, wait for a single clock cycles and store the outputs of the operations in the output registers. In figure 1(c) we depict a different implementation variant of the same computation, this time using nine registers and the same amount of adders and subtractors.

Summaries in 3 sentences.

that

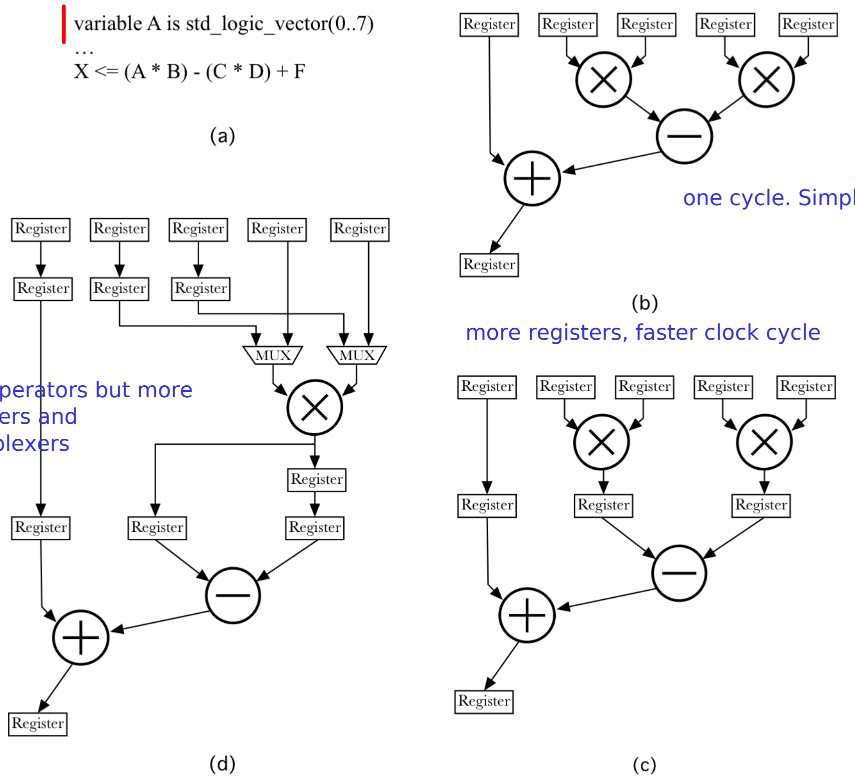
3 different possible hardware mapping of the same expression.

The increased number of registers allows for the circuit to be clocked at higher frequency as well as to be executed in a pipelined fashion. Finally, in figure 1(d) we depict yet another possible implementation of the same computation but using a single multiplier operator. This last version allows for the reuse in time of the multiplier operator and required thirteen register as well as multiplexers to route the inputs to the multiplier in two distinct control steps. As it is apparent, the reduction of number of operators, in this particular case the multipliers carries a penalty on increased number of registers, multiplexers [1] and increased complexity of the control scheme.

usefull?

```
variable A is std_logic_vector(0..7)
...
X <= (A * B) - (C * D) + F
```

(a)



one cycle. Simple

more registers, faster clock cycle

less operators but more registers and multiplexers

**Fig. 1.** Variants of mapping a computation to hardware: High-level source code (a); a simple hardware design (b); pipelined hardware design (c) design sharing resources (d).

This example illustrates the many facets of hardware design as exploited by high-level behavioral hardware synthesis. Behavioral synthesis techniques perform the

many degrees of freedom

[1] A $2 \times 1$ multiplexor is a combinatorial circuit with two data inputs, a single output and a control input, where the control input selects which of the two data inputs is transmitted to the output. It can be viewed as a hardware implementation of the C programming language selection operator : `out = (sel ? in1 : in2)`.

classical tasks of allocation, binding and scheduling of the various operations in a computation given specific target hardware resources. For instance, a designer can use behavioral synthesis tools (e.g., Mentor Graphics's Monet®) to automatically derive an implementation for a computation as expressed in the example in Figure 1(a) by declaring that it pretends to use a single adder and a single multiplier automatically deriving an implementation that resembles the one depicts in figure 1(d). The tool then derives the control scheme required to route the data from registers to the selected units as to meet the designers' goals. ~~This approach thus allows designs to retain some control of the generated hardware design while relieving them from the low-level and error prone details of hardware synthesis and thus have increased tremendously the productivity of designers over previous schematic capture and structural design approaches~~[2]

why do we call that "behavioral"

behavioral tools alows to (a) -> (d) by declaring the constraints

Despite the introduction of high-level behavioral synthesis techniques in commercially available tools, hardware synthesis and thus hardware compilation has never enjoyed the same level of success as traditional, high-level software, compilation. Sequential programming paradigms popularized by programming languages such as C/C++ and more recently by Java, allow programmers to easily reason about program behavior as a sequence of program memory state transitions. The underlying processors and the corresponding system-level implementations present a number of simple unified abstractions – such as a unified memory model, a stack, and a heap that do not exist (and often do not make sense) in customized hardware designs.

?

programming languages used for programmable processors underlie simple abstractions not adapted to custumized hardware design.

Hardware compilation, in contrast, has faced numerous obstacles that have impeded its progress and generality. When developing hardware solutions, designers must understand the concept of spatial concurrency (parallelism both in space and in time) that circuits do offer. Precise timing and synchronization between distinct hardware components are key abstractions in hardware. Solid and robust hardware design implies a detailed understanding of the precise timing of specific operations, including I/O, that simply cannot be expressed in language such as C, C++, or Java; for this reason, alternatives, such as SystemC [21] have emerged in recent years, which give the programmer considerably more control over these issues. The inherent complexity of hardware designs has hampered the development of robust synthesis tools that can offer high-level programming abstractions enjoyed by tools that target traditional architecture and software systems, thus substantially raising the barrier of entry for hardware designers in terms of productivity and robustness of the generated hardware solutions. At best, today hardware compilers can only handle certain subsets of mainstream high-level languages, and at worst, are limited to purely arithmetic sequences of operations with strong restrictions on control flow.

what is "parallelism in time"?

?

concurrency, synchronization, precise timing (operations, I/O, ...) => SystemC

too complex.
=> can handle subset of languages, restrictions on control flow

Nevertheless, the emergence of multi-core processing has led to the introduction of new parallel programming languages and parallel programming constructs that may be more amenable to hardware compilation than traditional languages and, similarly, abstractions such as a heap (e.g., pointer-based data structures) and a

new languages (designed for multicore) are more suitable for hadware

[2] Structural design approach require designers to explicitly implement all the control structures such as Finite-State-Machines (FSM) and the corresponding control signals for maximum design control and clock rate control.

unified memory address space are proving to be bottlenecks with respect to effective parallel programming. For example, MapReduce, originally introduced by Google to spread parallel jobs across clusters of servers [9], has been an effective programming model for FPGAs [29]; similarly, high-level languages based on parallel models of computation such as synchronous data flow [17], or functional single-assignment languages have also been shown to be good choices for hardware compilation [14, 13, 4].

map-reduce, synchronous data flow, functional single assignment good for hardware

Although the remainder of this chapter will be limited primarily to hardware compilation for imperative high-level programming languages with an obvious focus on SSA Form many of the emerging parallel languages will retain sequential constructs, such as control flow graphs, within a larger parallel framework. The extension of SSA Form, and SSA-like constructs, to these emerging languages, is an open area for future research; however, the fundamental uses of SSA Form for hardware compilation, as discussed in this chapter, are likely to remain generally useful.

focus on imperative languages still useful. SSA is good.

## 2 Why use SSA for Hardware Compilation?

Hardware compilation, unlike its software counterpart, offers a spatially oriented computational infrastructure that presents opportunities that can leverage information exposed by the SSA representation. We illustrate the direct connection between SSA representation form and hardware compilation using the mapping of a computation example in Figure 2(a). Here the value of a variable `v` depends on the control-flow of the computation as the temporary variable `t` can be assigned different values depending on the value of the `p` predicate. The representation of this computation is depicted in Figure 2(b) where a $\phi$-function is introduced to capture the two possible assignments to the temporary variable `t` in both control branches of the `if-then-else` construct. Lastly, we illustrate in Figure 2(c) the corresponding mapping to hardware.

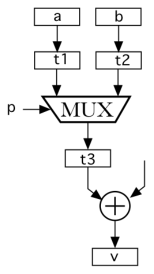
phi function == multiplexer on an example.

```
p = ... ;
if ( p ) then
  t = a;
else
  t = b;
end
v = t + ... ;
```

(a)

```
p = ... ;
if ( p ) then
  t1 = a;
else
  t2 = b;
end
t3 = Φ(t1,t2);
v = t3 + ... ;
```

(b)



(c)

**Fig. 2.** Basic hardware mapping using SSA representation.

The basic observation is that the confluence of values for a given program variable leads to the use of a $\phi$-function. This $\phi$-function abstraction thus corresponds in terms of hardware implementation of the insertion of a multiplexer logic circuit. This logic circuit uses the Boolean value of a control input to select which of its input's value is to be propagated to its output. The selection or control input of a multiplexer thus acts as a gated transfer of value that parallels the actions of an `if-then-else` construct in software.

Equally important in this mapping to hardware is the notion that the computation in hardware can now take a spatial dimension. In the suggested hardware circuit in Figure 2(c) the computation derived from the statement in both branches of the `if-then-else` construct can be evaluated concurrently by distinct logic circuits. After the evaluation of both circuits the multiplexer will define which set of values are used based on the value of its control input, in this case of the value of the computation associated with `p`.

exposes concurrency

In a sequential software execution environment, the predicate p would be evaluated first, and then either branches of the `if-then-else` construct would be evaluated, based on the value of p; as long as the register allocator is able to assign `t1`, `t2`, and `t3` to the same register, then the $\phi$-function is executed implicitly; if not, it is executed as a register-to-register copy.

The CFG adds implicit sequential order

There have been some efforts that could automatically convert the sequential program above into a semi-spatial representation that could obtain some speedup if executed on a VLIW type of processor. For example, `if-conversion` would convert the control dependency into a data dependency [1]: statements from the `if-` and `else` blocks could be interleaved, as long as they do not overwrite one another's values, and the proper result (the $\phi$-function) could be selected using a conditional-move instruction; however, in the worse case, this would effectively require the computation of both sides of the branch, rather than one, so it could actually lengthen the amount of time required to resolve the computation; in the spatial representation, in contrast, the correct result can be output as soon as two of the three inputs to the multiplexer are known (`s`, and one of `a` or `b`, depending on the value of `s`).

if conversion does the job but only partially

with lack of parallel ressources?

requires to be performed in //

When targeting a hardware platform, that advantage of the SSA representation is that assigning a value to each scalar variable exactly once makes the sequences of definitions and uses of each variable both formal and explicit. A spatially-oriented infrastructure can leverage this information to perform the computation in ways that would make no sense in a traditional processor. For example, in an FPGA, one can use multiple registers in space and in time to hold the same variable, and even simultaneously assign distinct values to it; then, based on the outcome of specific predicates in the program, the hardware implementation can select the appropriate register to hold the outcome of the computation. In fact, this is precisely what was done using a multiplexer in the example above.

?

A processor with ILP, it would

Hardware: fine grain parallelism of flexible/extensible level

hardware: flexibility to store/compute values where/when-ever we want

This example highlights the potential benefits of a SSA-based, when targeting hardware architectures that can easily exploit spatial computation, namely:

- Exposes the data dependences in each variable computation by explicitly incorporating into the representation each variable *def-use* chains. This allows a compiler to isolate the specific values and thus possible registers that conwould it be possible to refer to figure 1 to illustrate those points?

tribute to each of the assumed values of the variable. Using separate registers in hardware registers for disjoint live ranges, allows hardware generation to reduce the amount of resources in multiplexer circuits, for example. ??

– Exposes the potential for the sharing of hardware register not only in time and in space ?? providing insights for the high-level synthesis steps of allocation, binding and scheduling.
– Provides insight into control dependent regions where control circuitry is identical ?? and can thus be shared. This aspect has been so far neglected but might play an important role in the context of energy minimization.

While the Electronic Design Automation (EDA) community had for several decades now exploited similar information regarding data and control dependences for the generation of hardware circuits from increasingly higher-level representations (e.g., Behavioral HDL), SSA makes these dependences explicit in the intermediate representation itself. Similarly, the more classical compiler representations, using three-address instructions augmented with the *def-use* chains includes, in reality, the same information as the SSA-based representation. In the later however, and as we will explore in the next section, facilitates the mapping and selection of hardware resources.

Similar representation used in languages for HLS (eg Behavioral HDL).

SSA at IR level.

## 3 Mapping a Control Flow Graph to Hardware

We begin by describing the mapping to hardware of a computation expressed as a sequence of three-address instructions[3] in a basic block or a data-flow and then evolve by describing how to combine in the mapping the hardware circuits derived from multiple basic blocks, first in a straight-forward but inefficient fashion and then exploiting the knowledge captured in the SSA representation. In this section we are interested in hardware implementations or circuit that are spatial in nature and thus do not address the mapping to architectures such as VLIW [11] or Systolic Arrays [16]. ~~While these architectures pose very interesting and challenging issues, namely scheduling and resource use, we are more interested in exploring and highlighting the benefits of SSA representation which, we believe, are more naturally (although not exclusively) exposed in the context of spatial hardware computations.~~

### 3.1 Basic Block Mapping

As a basic block is a straight-line sequence of three-address instructions, a simple hardware mapping approach consists in composing or evaluating the operations in each instruction as a data-flow graph. The inputs and outputs of the instructions are transformed into registers[4] connected by nodes in the graph that

principle of a naive algo for BB mapping

~~[3] In this context we assume the use of a simple imperative three address instructions of the form `a = b ⊕ c` where `a`, `b`, and `c` are scalar variables or registers and ⊕ is a generic two input operation. Simple generalization for single-input operators and array operators are possible and follow the same rationale.~~

~~[4] As a first approach these registers are virtual and then after synthesis some of them are materialized to physical registers in a process similar to register allocation in software-oriented compilation.~~

represent the operators. ~~Figure 3 outline an algorithm that creates such a data-flow~~ representation of the computation amenable to direct hardware synthesis. The algorithm sequentially evaluates each instruction, in a symbolical fashion maintaining a binding of the scalar variable names to a register in the data-flow representation using the function `mapNameToRegister`. Whenever no binding is found the algorithm uses the function `allocateRegister` to create a new node in the data-flow representation that captures a value of a variable and thus of a partial computation. To assign the new value of a variable to a new hardware register the algorithm uses the `bindNameToRegister` function. It then uses the function `createComputationNode` to create a representation of the computation with the registers derived previously.

```
for each instruction i: z = x ⊕ y in basic block B do
 rx = mapNameToRegister(x);
 if(rx = nil) then
  rx = allocateRegister(x);
 end if
 ry = mapNameToRegister(y);
 if(ry = nil) then
  ry = allocateRegister(y);
 end if
 rz = allocateRegister(z);
 bindNameToRegister(rz, z);
 createComputationNode(rz, rx, ⊕, ry);
end for

for all variables v in basic block B do
 if (v is live at output of B) then
  saveRegister(mapNametoRegister(v));
 end if
end for
```

Figure c with registers at the level between the - & +.

**Fig. 3.** Basic-Block hardware mapping algorithm.

As a last step the algorithm saves the registers corresponding to the live variables at the output of the basic block. These registers will be used to communicate the values of the live variables to subsequent hardware circuits, either by having their values copied to a local storage or ~~directly connected via wires to other hardware circuits (see description in section 3.2)~~. As a result of the "evaluation" of the instructions in the basic block this algorithm constructs a hardware circuit that has as input registers that will hold the values of the input variables to the various instructions and will have as outputs registers that hold only variables that are live outside the basic block.

registers for live-in
registers for live-out
registers for live variables between each instruction.

## 3.2 Basic Control-Flow-Graph Mapping

One can combined the various hardware circuits corresponding to a control-flow graph in two basic approaches, respectively, *spatial* and *temporal*. The spatial

How do we express that a BB is not executed (branch) and that we should take the result from one branch and not the other (multiplexing)?
There is a bolean value at every program point that propagates?

form of combining the hardware circuits consists in laying out the various circuits spatially by connecting variables that are live at the output of a basic block (and therefore corresponding to output registers of the corresponding hardware circuit) to the registers that will hold the values of those same variables in a subsequent hardware circuit corresponds to basic blocks that execute in sequence [14, 15].

spatial: interconnect variables between BB

In the temporal approach the hardware circuits corresponding to the various CFG basic blocks are not directly interconnected. Instead their input and output registers are connected via dedicated buses to a local storage module. An execution controller äctivatesä basic block or a set of basic blocks by transferring data between the storage and the input registers of the hardware circuits to be activated. Upon execution completion the controller transfer the data from the output registers of each hardware circuit to storage. These data transfers need not necessarily be carried out sequentially but instead can leverage the aggregation of the outputs of each hardware circuits to reduce transfer time to and from storage via dedicated wide buses.

temporal: bus to storage module + controller

output registers are filled at the same time in the previous solution for BB mapping. This is more general here?

This is already SSA?

?

- difference between dedicated bus and register/wire?
- if a bus is shared by several BB, how do you decide which BB share the same bus?

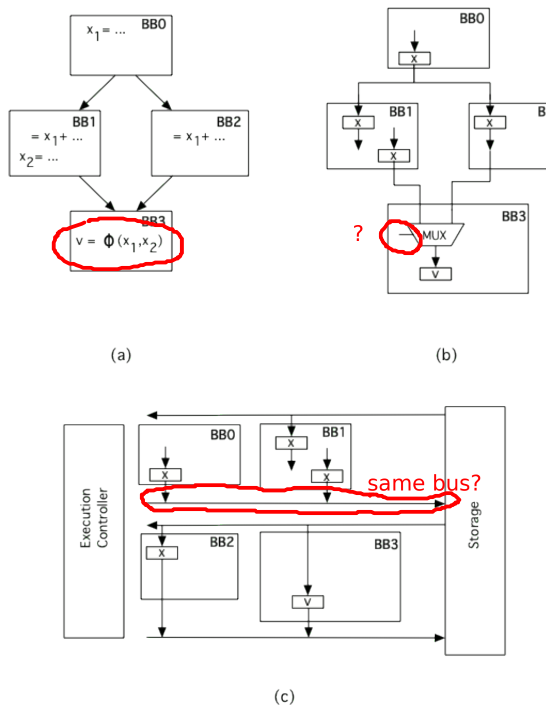


**Fig. 4.** Combination of hardware circuits for multiple basic blocks. CFG representation (a); Spatial mapping (b); Temporal mapping (c).

Again, this is very general here where different BBs can share some ressources...

The temporal approach describe above is well suited for the scenario where the target architecture does not have sufficient resources to simultaneously implement the hardware circuits corresponding to all basic blocks of interest as it trade offs why? bus comm. takes time?

execution time for hardware resources.

In such a scenario, where the hardware resources are very limited or the hardware circuit corresponding to a set of basic blocks is exceedingly large, one could opt for partitioning a basic block or set of basic block into smaller blocks until the space constraints for the realization of each hardware circuit are met.[5] The computation thus proceed as described above by saving the values of the output registers of the hardware circuit corresponding to each smaller block.

split into small blocks & share common ones

These two approaches, illustrated in Figure 4, can obviously be merged in a hybrid implementation, lead to distinct control schemes for the orchestration of the execution of computation in hardware and their choice depends heavily on the nature and granularity of the target hardware architecture. For fine-grain hardware architectures such as FPGAs a spatial mapping can be favored, for coarse-grain architectures a temporal mapping is common [6].

spatial versus temporal: depends on the "target" architecture

why?
programmable architectures?
fine/coarse grain architectures?

I do not see why

what do you mean exactly by pipelining execution for hardware

I see why it is area inefficient but I do not see the realtionship with the fact that most of the time it exposes no parallelism between BBs.

temporal?

While the overall execution control for the temporal mapping approach is simpler, as it transfers to and from storage of the variables upon the transfer of control between hardware circuits, a spatial mapping approach makes it more amenable to take advantage of pipelining execution techniques and speculation [6]. The temporal mapping approach can be, however, area-inefficient, as at often only one basic block will execute at any point in time. This issue can, nevertheless, be mitigated by exposing additional amounts of instruction-level parallelism by merging multiple basic blocks into a single *hyper-block* and combining this aggregation with loop unrolling. Still, as these transformations and their combination, can lead to a substantial increase of the required hardware resources, a compiler can exploit resource sharing between the hardware units corresponding to distinct based blocks to reduce the pressure on resource requirements and thus lead to feasible hardware implementation designs (see e.g. [19]). As these optimizations are not specific to the SSA representation, we will not discuss them further here.

spatial => exposes ILP but space inefficient

=> mixed solution of temporal and spatial

## 3.3 Control-Flow-Graph Mapping using SSA

In the case of the spatial mapping approach (described in section 3.2 the SSA form plays an important role in the minimization of multiplexers and thus in the simplification of the corresponding data-path logic and execution control.

Consider the illustrative example in Figure 5(a). Here basic block `BB0` defines a value for the variables `x` and `y`. One of the two subsequent basic blocks `BB1` redefines the value of `x` whereas the other basic block `B2` only reads them.

avoid notes. Include it into the text.

[5] In reality this is the approach in every processor today. It limits the hardware resources to the resources requires for each of the ISA instructions and schedules them in time at each step saving the state (registers) that were the output of the previous instruction.

[6] Speculation is also possible in the temporal mode by activating the inputs and execution of multiple hardware blocks and is only limited by the available storage bandwidth to restore the input context in each block which in the spatial approach is trivial.

A naive implementation based exclusively on *live* variable analysis would use for both variables x and y multiplexers to merge their values as inputs to the hardware circuit implementing basic block BB3 as depicted in Figure 5(b). As can be observed, however, the SSA-form representation captures the fact that such a multiplexer is only required for variable x. The value for the variable y ca be propagated either from the output value in the hardware circuit for basic block BB0 (as shown in Figure 5(c)) or from any other register that has a valid copy of the y variable. The direct flow of the single definition point to all its uses, across the hardware circuits corresponding to the various basic blocks in the SSA form thus allows a compiler to use the minimal number of multiplexer strictly required[7].

def-use chains (with unique def)
instead of control flow
=> less multiplexers

pruned minimal SSA provides
minimum # of mulitplexers

minimal pruned ?

this is still not clear for me how values "live" in the circuit. Can a value stay in a register during several consecutive cycles?
How do you know when you can use/free it...

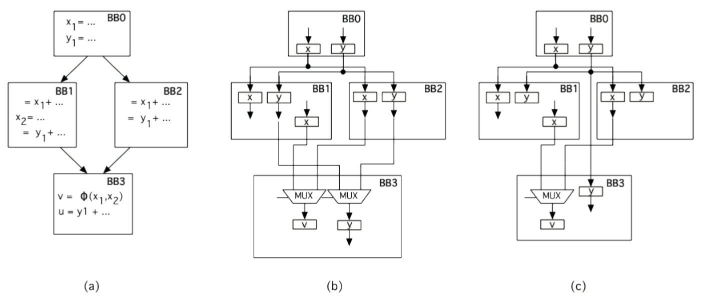


**Fig. 5.** Mapping of variable values across hardware circuit using spatial mapping; CFG in SSA form (a); naive hardware circuit using live variable information for multiplexer placement (b); hardware circuit using $\phi$-function for multiplexer placement (c).

An important aspect regarding the implementation of a multiplexer associated with a $\phi$-function is the definition and evaluation of the predicate associated with each multiplexer's control (or selection) input signal. In the basic SSA representation the selection predicates are not explicitly defined, as the execution of each $\phi$-function is implicit when the control-flow reaches it. When mapping a computation the $\phi$-function to hardware, however, one has to clearly define which predicate to use to be included part of the hardware logic circuit that defines the value of the multiplexer circuit's selection input signal. The Gated-SSA form (see e.g. [26]) of the SSA form explicitly captures the necessary information in the representation. As the names associated with each value correspond to the precise value (with the correct number index) used in the predicate evaluation this variant of the SSA form is particularly useful for hardware synthesis. The generation of the hardware circuit simply has to use the register's value ~~(possibly an intermediate value that correspond to the name of each of the variable referenced in the predicate expression of this Gated SSA form~~. Figure 6 illustrates an example of a mapping using the information gated-SSA form.

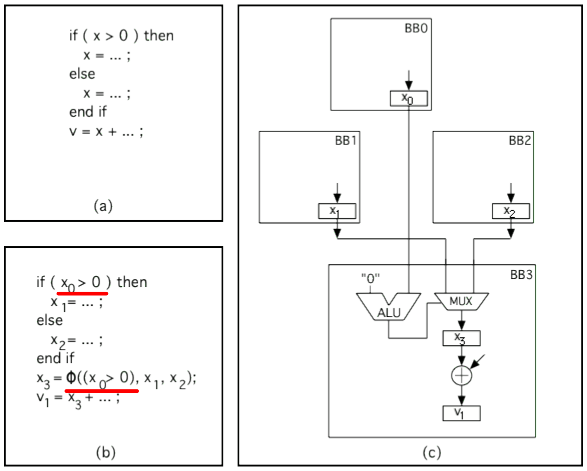
implementation of a multiplexer
Gated SSA provides the predicate

gated SSA has predicate expressions not directly predicate registers?

did not understood
so probably not important :-)

[7] Under the scenarios of a spatial mapping and with the common disclaimers about static control-flow analysis.

the predicate expression
is computed twice?
why is it better than having a predicate
saying if a BB is executing or not?



```
if ( x > 0 ) then
   x = ... ;
else
   x = ... ;
end if
v = x + ... ;
```

(a)

```
if ( x0 > 0 ) then
  x1 = ... ;
else
  x2 = ... ;
end if
x3 = Φ((x0 > 0), x1, x2);
v1 = x3 + ... ;
```

(b)

**Fig. 6.** Hardware generation example using Gated-SSA form; original code (a); Gated-SSA representation (b); hardware circuit implementation using spatial mapping (c).

it seems important to have a description of PDG (not necessary its consutruction) somewhere in the book.

When combining multiple predicates in the Gated-SSA form it is often desirable to leverage the control-flow representation in the form of the Program-Dependence-Graph (PDG) [10]. In this representation regions of the code, and hence basic blocks, that shared common execution predicates (i.e., both execute under the same predicate conditions) are linked to the same *region* nodes in this representation. Nested execution conditions are easily recognized as the corresponding nodes are hierarchically organized in the PDG representation. As such, when generating code for a given basic block an algorithm will examine the various region nodes associated with a given basic block and compose (using AND operators) the outputs of the logic circuits that implementation the predicates associated with these nodes. If ~~no~~ hardware circuit already exists for the same predicate, the implementation simply reuses its output signal. Otherwise, creates a new hardware circuit. This lazy code generation and predicate composition achieves the goal of hardware circuit sharing as illustrated by the example in Figure 7 where some of the details were omitted for simplicity.

PDG allows to link an entire region to its corresponding control points it depends on.

PDG exposes reuse of predicates

## 3.4 $\phi$-Function and Multiplexer Optimizations

We now describe a set of hardware-oriented transformations that can be applied to improve the amount of hardware resources devoted to multiplexer implementation. Although these transformations are not specific the mapping of computations to hardware in SSA form, the explicit representation of the selection construct in SSA makes it very natural to map and therefore manipulate and transform the resulting hardware circuit using multiplexer. Other operations in the intermediate representation (e.g., ~~conditional moves,~~ predicated instructions) can also yield multiplexers in hardware without the explicit use of SSA Form.

rewrite this sentence

what are 7(a), 7(b) respectively?

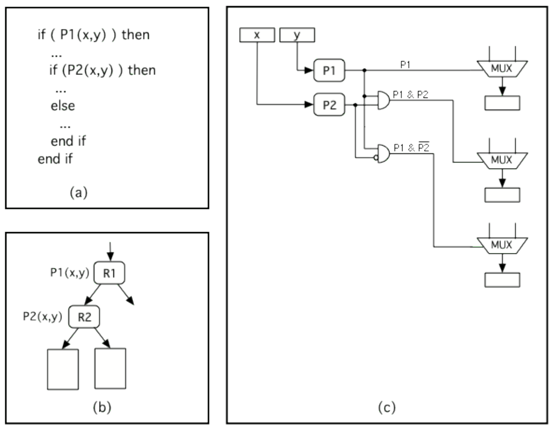


**Fig. 7.** Use of the predicates in region nodes of the PDG for mapping into the multiplexers associated with each $\phi$-function.

A first transforation is motivated by a well-known result in computer arithmetic: integer addition scales with the number of operands. Building a large and unified k-input integer addition circuit is more efficient than adding k integers two at a time [28, 22]. Moreover, hardware multipliers naturally contain multi-operand adders as building blocks: a partial product generator (a layer of AND gates) is followed by a multi-operand adder called a *partial product reduction tree*. For these reasons, there have been several efforts in recent years to apply high-level algebraic transformations to application data flow with the goal of merging disparate addition operations with one another and with the partial product reduction trees of multiplication operations. The basic flavor of these transformations is to push the addition operators toward the outputs of a data flow graph, so that they can be merged at the bottom. Example of these transformations that use multiplexers are depicted in Figure 8(a,b). Figure 8(c) depicts a similar transformation that merges two multiplexers sharing a common input, while exploiting the commutative property of the addition operator [27].

useful to regroup additions so as to merge them.

i do not understand: a partial product generates 1 result. So why a multi operand adder?

?

not clear

where doe the multiplexers come from? the phi? If yes, what is the advantage of moving an addition (that might not be executed) toward a BB in which it will necessarily be executed?

A second transformation that can be applied to multiplexers is specific FPGAs whose basic building block consists of a k-input lookup table (LUT) logic element. which can be programmed to implement any k-input logic function. For example, a 3-input LUT (3-LUT) can be programmed to implement a multiplexer with two data inputs and one selection bit; similarly, a 6-LUT can be programmed

FPGA made of k-LUT usually k=4

combining multiplexers

how SSA can help?

to implement a multiplexer with four data inputs and two selection bits. In particular, many FPGAs devices are organized using 4-LUTs, which are too small to implement a multiplexer with four data inputs, but leave one input unused when implementing multiplexers with two data inputs. This features can be explored to reduce the number of 4-LUTs required to implement a tree of multiplexers [20].

how to you do that?
how SSA helps?

BB boundary?

idem

we do not see how SSA helps

examples of addition moving across a multiplexer

merging multiplexers to reduce there numbers

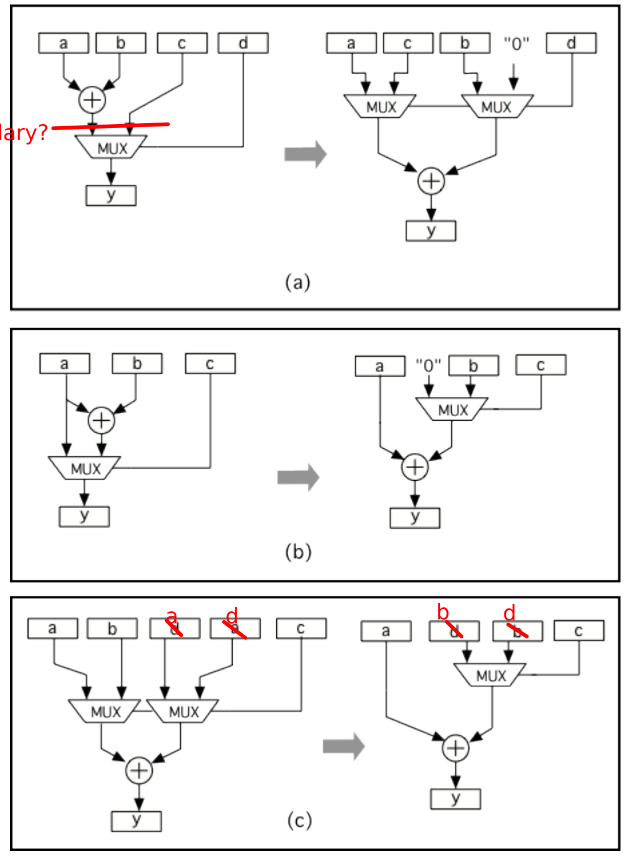


**Fig. 8.** Multiplexer-Operator transformations: juxtapose the positions of a multiplexer and an adder (a,b); reducing the number of multiplexers placed on the input of an adder (c).

## 3.5 Implications of using SSA-form in Floor-planing

For spatial oriented hardware circuits moving a $\phi$-function from one basic block to another can alter the length of the wires that are required to transmit data from the hardware circuits corresponding to the various basic blocks and thus have a substantial impact on the overall maximum hardware design clock rate. We illustrate this effect via an example as depicted in Figure 9 [15]. In this figure each

why wires length have an impact on clock rate?

wires length have an impact. On opt. of floor planing is to reduce wires length.

basic block is mapped to ~~is~~ a distinct hardware unit, whose spatial implementation is approximated by a rectangle. A floor-planning algorithm must place each of the units in a two-dimensional plane while ensuring that no two units overlap. 5 As can be seen in Figure 9(a) placing the block D on the right-hand-side of the plane will results in several mid-range and one long-range wire connections. However, placing block D at the center of the design will virtually eliminate all mid-range connections as all connections corresponding to the transmission of the values for variable x are now next-neighboring connections.

BB mapping

how a wire can cross another one?

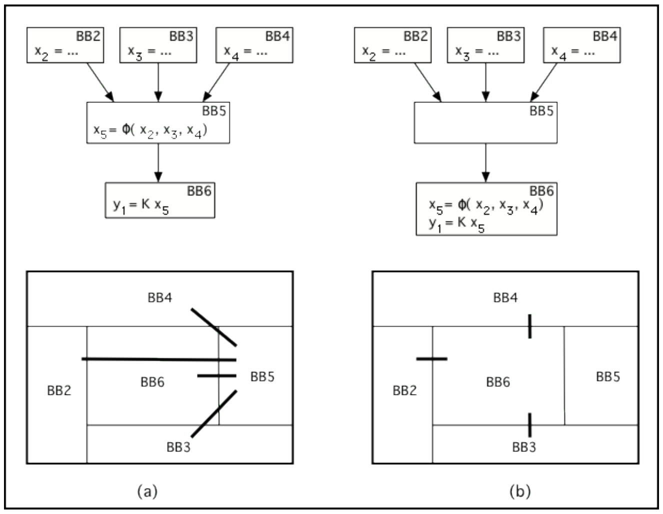


**Fig. 9.** Example of the impact of $\phi$-function movement in reducing hardware wire length.

As illustrated by this example, moving a multiplexer from one hardware unit to another can significantly change the dimensions of the resulting unit, which is not under the control of the compiler. Changing the dimensions of the hardware units fundamentally changes the placement of modules, so it is very difficult to predict whether moving a $\phi$-function will actually be beneficial. For this reason, compiler optimizations that attempt to improve the physical layout must be performed using a feedback loop so that the results of the lower-level CAD tools that produce the layout can be reported back to the compiler.

phi-function moving

what means moving a phi-function?
for gated SSA is seems clear
otherwise maybe Lennart's chapter
(semantic) provide an answer
to this kind of question.

## 4 Existing SSA-based Hardware Compilation Efforts

Several research projects have relied on SSA-based intermediate representation that leverage control- and data-flow information to exploit fine grain parallelism. Often, but not always, these efforts have been geared towards mapping computations to fine-grain hardware structure such as the ones offered by FPGAs.

The standard approach followed in these compilers has been to translate a high-level programming language such as Java in the case of the Sea Cucumber [25] compiler or C in the case of the ROCCC [12] to sequences of intermediate instructions. These sequences are then organized in basic blocks that compose the control-flow graph (CFG). For each basic block a data-flow-graph (DFG) is typically extracted followed by conversion in to SSA representation, possibly using predicates associated with the control-flow in the CFG thus explicitly using Predicated SSA representation (PSSA [7, 8, 23]).

As an approach to increase the potential amount of exploitable instruction-level parallelism (ILP) at the instruction level, many of these efforts (as well as others such as the earlier Garp compiler [5]) restructure the CFG into hyper-blocks [18]. An hyper-block consists on a single-entry multi-exit regions derived from the aggregation of multiple basic blocks thus serializing longer sequences of instruction. As not all instructions are executed in a hyper-block (due to early exit of a block), hardware circuit implementation must rely on predication to exploit the potential for additional ILP.

can you say anything about SSI for hardware synthesis.

The CASH compiler [3] uses an augmented predicated SSA representation with tokens to explicitly express synchronization and handle *may-dependences* thus supporting speculative execution. This fine-grain synchronization mechanism is also used to serialize the execution of consecutive hyper-blocks, thus greatly simplifying the code generation.

This citation is to make chapters without citations build without error. Please ignore it: [**?**].

## References

1. J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proc. of the ACM Conf. on Principles of Programming Languages (POPL)*, pages 177–189. ACM Press, 1983.
2. P. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
3. M. Budiu and S. Goldstein. Compiling application-specific hardware. In *Proc. of the Intl. Conf. on Field Programmable Logic and Applications (FPL)*, pages 853–863, 2002.
4. W. Bhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a single assignment programming language to reconfigurable systems. *The Journal of Supercomputing*, 21(2):117–130, February 2002.
5. T. Callahan, J. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, 2000.
6. J. Cardoso and M. Weinhardt. XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture. In *Proc. of the 12th Intl. Conf. on Field-Programmable Logic and Applications (FPL'02)*, pages 864–874, London, UK, 2002. Springer-Verlag.
7. L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proc. of the 1999 Intl. Conf. on Parallel Architectures*

*and Compilation Techniques (PACT'99)*, page 245, Washington, DC, USA, 1999. IEEE Computer Society.

8. François de Ferrière. Improvements to the psi-SSA representation. In *Proc. of the 10th Intl. Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 111–121, 2007.
9. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
10. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, 1987.
11. J. Fisher, P. Faraboschi, and C. Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier Science Publishers B. V., 2005.
12. Z. Guo, B. Buyukkurt, J. Cortes, A. Mitra, and W. Najjar. A compiler intermediate representation for reconfigurable fabrics. *Int. J. Parallel Program.*, 36(5):493–520, 2008.
13. A. Hagiescu, W.-F. Wong, D. Bacon, and R. Rabbah. A computing origami: folding streams in FPGAs. In *Proc. of the 2009 ACM/IEEE Design Automation Conf. (DAC)*, pages 282–287, 2009.
14. A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah. Optimus: Efficient Realization of Streaming Applications on FPGAs. In *Proc. of the 2008 Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 41–50, 2008.
15. A. Kaplan, P. Brisk, and R. Kastner. Data communication estimation and reduction for reconfigurable systems. In *Proc. of the 40th Design Automation Conf. (DAC'03)*, pages 616–621, 2003.
16. M. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, Norwell, MA, USA, 1989.
17. E. Lee and D. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, 75(9):1235–1245, 1987.
18. S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proc. of the 25th Annual Intl. Symp. on Microarchitecture (MICRO)*, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
19. S. Memik, G. Memik, R. Jafari, and E. Kursun. Global resource sharing for synthesis of control data flow graphs on FPGAs. In *Proc. of the 40th Design Automation Conf. (DAC'03)*, pages 604–609, 2003.
20. P. Metzgen and D. Nancekievill. Multiplexer restructuring for FPGA implementation cost reduction. In *Proc. of the ACM/IEEE Design Automation Conf. (DAC)*, pages 421–426, 2005.
21. P. Panda. SystemC: a modeling platform supporting multiple design abstractions. In *Proc. of the 14th Intl. Symp. on Systems Synthesis (ISSS'01)*, pages 75–80, New York, NY, USA, 2001. ACM.
22. P. Stelling, C. Martel, V. Oklobdzija, and R. Ravi. Optimal circuits for parallel multipliers. *IEEE Trans. on Computers*, 47(3):273–285, March 1998.
23. A. Stoutchinin and F. de Ferrière. Efficient static single assignment form for predication. In *Proc. of the 34th Annual Intl. Symp. on Microarchitecture (MICRO)*, pages 172–181, Dec. 2001.
24. D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

25. J. Tripp, P. Jackson, and B. Hutchings. Sea Cucumber: A Synthesizing Compiler for FPGAs. In *Proc. of the 12th Intl. Conf. on Field-Programmable Logic and Applications (FPL'02)*, pages 875–885, London, UK, 2002. Springer-Verlag.
26. P. Tu and D. Padua. Gated SSA-based demand driven symbolic analysis for parallelizing compilers. In *Proc. of the ACM Intl. Conf. on Supercomputing (SC'95)*, pages 414–423, New York, NY, USA, 1995. ACM.
27. A. Verma, P. Brisk, and P. Ienne. Dataflow transformations to maximize the use of carry-save representation in arithmetic circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1761–1774, October 2008.
28. C.S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. on Electronic Computers*, 13(14):14–17, February 1964.
29. J. Yeung, C. Tsang, K. Tsoi, B. Kwan, C. Cheung, A. Chan, and P. Leong. Map-reduce as a programming model for custom computing machines. In *Proc. of the 2008 16th Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM'08)*, pages 149–159, Washington, DC, USA, 2008. IEEE Computer Society.