Java Byte Code Synthesis for Reconfigurable Computing Platforms

Christophe Dubach

Computer Science, master thesis

Processor Architecture Laboratory

Professor : Paolo Ienne

Assistant : Miljan Vuletic

Jan 2005

* updated version *

Foreword

As you have probably noticed in the title page, this version of this report is an updated version, relative to the official one. It contains some updates concerning the performance comparison of the IDEA coprocessor and other minor fixes.

Nevertheless, the content is almost the same as the official one. The main purpose of this updated version is to provide more accurate results about the IDEA coprocessor, used as example to get a view of the compiler performance.

Contents

Chapter 1

Introduction

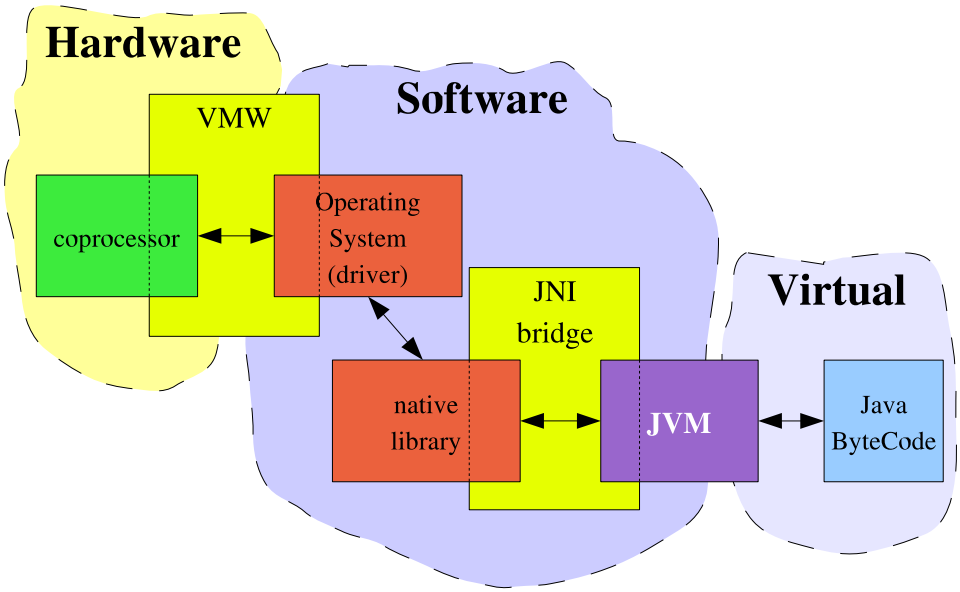


Figure 1.1: General picture of the system. The virtual part is composed of the Java program. It will be executed by the JVM. By using the native library (JNI bridge) the JVM will access the VMW interface driver in the software part. Then the coprocessor is executed ; this is the hardware part.

The JBC (Java Byte Code) is a virtual assembly code that executes in a virtual machine ; the JVM (Java Virtual Machine). While the first virtual machines did only emulation, nowadays virtual machines do *ahead-of-time* [1] and *just-in-time* [2] compilation of the JBC into native assembly code.

This project deals with the synthesis of hardware *ahead-of-time*. It is clear that the main objective would be to do hardware synthesis *just-in-time*. The task is thus to write a compiler that takes JBC as input and produces VHDL code. The usual synthesis tools such as Quartus are then fed with this VHDL code to produce hardware.

The big picture is presented in Figure 1.1. The objective of this project is to start with the "virtual" part of the figure and to generate the "hardware" part. The interface between the "software" part and the "virtual" part has been the purpose of a previous project. This is described in the next section. The interface between "hardware" and "software" side is described in Section 1.2 and has been the subject of a PhD.

1.1 Transparent use of a coprocessor from Java

In a previous semester project we have developed ([1], [2]) a thin library that allows unmodified Java program to use a coprocessor. In addition, this library is independent from the JVM, since it use JNI (Java Native Interface) which is part of the Java platform and thus runs on any platform.

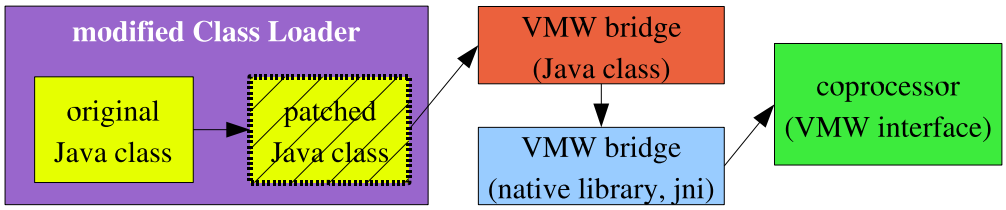


Figure 1.2: From a Java class to its execution in the coprocessor. A modified class loader patch the original Java class to relay the call of the method to be run in the coprocessor, in a method of the bridge. Then this bridge launches the coprocessor.

There are two components ; the native library that does the bridge between Java and the operating system, and a modified Class Loader[3], which

[1] In *ahead-of-time* mode, the compilation is done before the execution of the code.

[2] In *just-in-time* mode, the compilation is done when required, that is during the program execution.

[3] The Class Loader is responsible for loading each class of an application in the JVM. It parses the class and creates a corresponding Object in the JVM.

actually "patches" the method to be executed on the coprocessor, in order for it, to do a call to the native library that will launch the coprocessor, instead of calling its Java implementation.

Figure 1.2 shows the typical steps that are required to execute a Java method in the coprocessor. First, the Java class containing the method to be executed in the coprocessor is patched by the modified Class Loader. Then, when we do a call to the method, it is redirected by a call to a method of a Java class that acts as a bridge. The method of this last class calls the native method of the native library, which finally executes the coprocessor.

1.2 VMW interface

In order to simplify the development of this project, we have used the VMW (Virtual Memory Window) [3] developed at the LAP (Processor Architecture Laboratory) of EPFL (Swiss Federal Institute of Technology, Lausanne). This interface permits to access the memory with user-space addresses. It acts as a MMU (Memory Management Unit) with a cache.

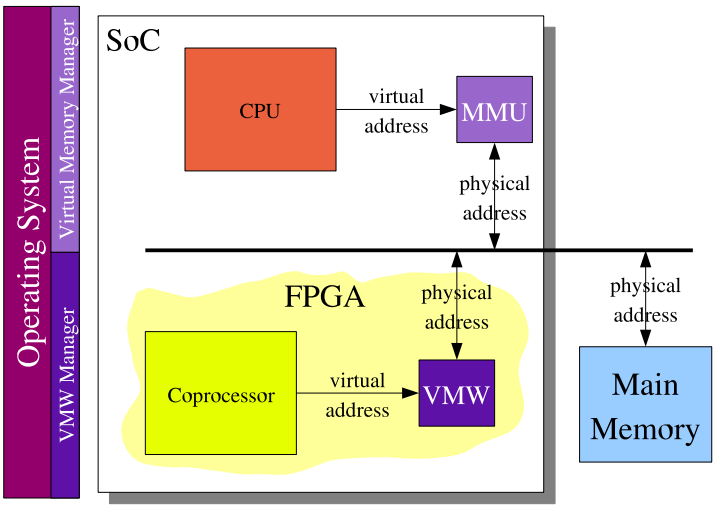


Figure 1.3: A schema of our hardware system. We have in the same system the main CPU which access the memory through an MMU. And in the other part of the chip, we have the coprocessor (in the FPGA) that access the memory with its VMW interface. Both use user-space addresses.

Figure 1.3 shows this VMW in the hardware system (which is described in the next section).

1.3 Technical details of the system

Now that we have seen the key-components of the system, let's see the rest and the technical details.

We use a small development board also developed at the LAP of EPFL. This board use an Altera EPXA integrated circuit that contains an "hard-core" ARM922T processor (which runs at 133 MHz)and an FPGA. It is equipped with 64MB of RAM and 8MB of flash memory. This EPXA circuit constitutes a SoC (System-on-Chip) and it is also possible to have a NIOS soft-core processor on the same chip, but we have not used this feature at all.

On the software side, we use a Linux kernel[4] (2.6.0-test11). All the standard UNIX commands are presents, thanks to BusyBox (1.00pre-4) [5]. An"open Source" JVM, called Kaffe[6] is used (version 1.1.3). The tool-chain, that is used to do cross-compiling is GCC[7] (3.3.2).

Although new versions of those softwares are now out, wisdom tells us not to upgrade to newer versions (it has taken one semester project to put the system together), as the system we have is fully working.

On the compiler side (the compiler created in this project), we use Java (version 1.4) and the BCEL library [8] (Byte Code Engineering Library). Quartus 2 (version 4.1) is used to do the synthesis of VHDL code.

[4]http://www.kernel.org/
[5]http://www.busybox.net/
[6]http://www.kaffe.org
[7]http://www.gnu.org/software/gcc
[8]http://jakarta.apache.org/bcel/

Chapter 2

Operations and resources associated with Java Byte-Code

We define a model for the byte-code. First we have mapped the different instructions to operations that can take place in hardware. Notice that not every instructions have a corresponding operation in hardware.

Different properties are then associated with the operations. Those properties help to support an hardware execution that gives exactly the same results (data results, world state) as in software execution. But let's first see what JBC is.

2.1 What is Java Byte-Code ?

JBC is a virtual assembly code, in the sense that it is executed by a virtual machine , the Java Virtual Machine (JVM). Its architecture is of type Load/Store, where each instruction does not directly access the memory, but dedicated instructions are used to do so. Access to the memory is provided by the *ASTORE* and *ALOAD* instruction family, that are used to access arrays (*ASTORE* means array store and *ALOAD* array load). Their exists one *ASTORE* and *ALOAD* instruction per Java type (byte, char, short, int, long, float and double). For instance, the load operation for byte array is called *BALOAD*.

The result of a JBC instruction is always stored on a virtual stack. Special instructions exist to manipulate this stack. But as we will see, in our case this stack will not appear on the hardware side. Otherwise, it also provides a kind of register abstraction by the means of local variables. Local variables are used to pass parameters and to store the results of the variables declared in the Java source code (this is not required, but in general there is a direct

mapping between source code variables and local variables in the JBC).

Each method has its own stack and local variable pool. It is impossible to modify them from outside of the method, except by the mean of method call, which actually passes the parameters directly in the local variable pool of the called method. The first parameter is stored in local variable 0, the second in local variable 1 and so on. The called method then return the result by calling the RETURN instruction, which pushes on the stack of the caller, the result. The stack and the local variable pool are called a *frame*. Each method call create a new *frame* and the content of this *frame* is only accessible by the owner of it (the method that is executed).

Another particularity of the JVM is the fact that it works only with integer type. All operation produces results on 32 bits ; the local variables pool and the stack accept only 32 bits values. Long and double values (which are 64 bits width) take two places on the stack and two slots when stored in local variable pool. In fact some instructions produce a result on 64 bits, but we can see them as if they produce two results of 32 bits. In our compiler, we have only considered instructions that act on 32 bits, and we have left out (for the sake of simplicity) all instructions that deal with long and double type. But it would not be quite difficult to extend our compiler to support those instructions.

More information about JBC can be found in [4].

2.2 Classes of operation

First let's see the difference between instructions and operations. In our report, the term instruction will always refer to JBC instruction. Operation will refer as an abstract view of the instruction ; there is no more stack or local variable information, but only a kind of function that takes an known number of inputs (that can be 0) and produces (sometime) an output.

There is two main classes[1] of operation. The distinction between those two classes is done according to the delay of the operations. We define the delay of an operation as the number of clock cycle needed to produce the result or to perform an action (such as memory write operation). The two classes are simply the class of operations whose delay is known (*data-independent delay*) and the other class the one whose delay is unknown (*data-dependent delay*, also called *unbounded delay* operation in this report).

All the operations that belong to the first class execute in a known constant time. Such operations are for instance addition, multiplication and so

[1]Those classes have nothing to do with OOP (Object Oriented Programming) classes.

on. The other class has an unknown delay, we find memory operations (write or read) and the CALL operation, which does method call.

2.3 Properties of the operations

We define two boolean proprieties for each operation. The first one, Access-World, is true when an operation reads a data (or does a synchronisation task) with the world. Memory read operation has for instance this propriety. The second is Modify-World. It determines that the operation will write a data to the world. For instance, memory write operation, has this propriety. Some operations could have both proprieties, for instance in a case of a function call operation.

Concretely the Modify-World operation can be seen as a set that contains the Access-World one. The Modify-World property is stronger than the Access-World one. Every operations that have this property must be executed in the same order as the original JBC, in order to be conservative with the memory (and not only the memory, but also the rest of the world). The Access-World property is less stronger, because consecutive 'Access-World operations can be executed in any order, as soon as there is no Modify-World operation in between two Access-World operations.

2.4 Resource

The different operations are "implemented" in hardware as resources. As we target an FPGA, resources are implemented with LE (*logic elements*). An FPGA is an array of LE (aka cells).

Those resources are real piece of hardware, which can have their own state. For instance the resource that is responsible for doing memory access has an internal controller.

Each operation is mapped to one resource. In our compiler, we have chosen, for the sake of simplicity, to support only one resource per operation, but a resource can be used by more than one operation.

For each resource we define its different properties :

- the delay, which is the number of clock cycle needed to finish its execution once started
- the name of the input signals
- the available number

- and a flag that indicated if the resource is allowed to be shared or not

An important point to respect is the fact that, at any time, we are allowed to execute as many as available resources. In other words, the use of one resource should not influence the use of another one. Consider the problem of memory access. We have two kind of memory access ; read and write. In order to guaranty this rule, the resource that will "implement" the memory read and the memory write operation has to be the same. That's why we have to add another property to the resource : the function. The function is in fact a selection signal that will inform the resource about the actual operation we want to execute. If we take the example of an ALU (Arithmetic and Logic Unit), there is a function that defines if we want to do an addition, an OR or an AND for instance.

In our case, we have only one resource that is responsible of the communication with the VMW interface. This resource is thus able to do memory read or write operation, read the parameters of the method, return a value and so on. If we would have had two resources, one for doing the memory read operation and the other for the memory write operation, it would have violated the rule that at any time we are allowed to execute as many as available resource, since two resources would have tried to access simultaneous the VMW interface (which is not allowed).

2.4.1 Resource contract

The implementation of the resources is not free (as in beer), but has to ensure some rules. Those rules will guarantee an exact behaviour of the generated code. Here are the rules to respect :

1. the result of the resource should be valid till the next execution of the resource
2. the resource should be registered ; at least one clock cycle is needed to produce the result
   (a) this involves the need of a clock and reset signal for each resource
   (b) the needs of an enable signal

As we will see later, those rules will allow us to occupy more efficiently the area in the FPGA. For instance, the guarantee that the result of a resource is valid till its next execution will allow us to avoid the use of additional registers, in order to save the result that will be used later on by another resource.

2.4.2 Resource implementation

Once the contract is respected, the implementation of the different resources is free. The resource implementation resides outside the compiler. This gives a big flexibility to test different configurations. On the compiler side, a configuration file "informs" the compiler about the resources. This configuration file gives for each operation the resource that implements it. It also gives detailed information about the resource itself, such as its delay, its available number, if it is possible to share the resource and even the different functions it can execute.

Listing 2.1: Example of xml-based configuration file. This file is used by the compiler to get information about operations and the corresponding resources.

```
<Config>
 <Resources>
    ...
   <Resource name="add_sub" number="-1">
    <DataIn num="0" name="in1" width="32"    ...>
    <DataIn num="1" name="in2" width="32"    ...>
    <DataOut num="0" name="out1" width="32" ...>
    <Select name="funSel" width="1">
     <Function name="add" delay="1" select="1"/>
     <Function name="sub" delay="1" select="0"/>
    </Select>
    <Enable name="enable"/>
    <Clock name="clk"/>
    <Reset name="reset"/>
  </Resource>
   ...
 </Resources>

 <Operations>
  ...
 <Operation name="IADD">
  <Resource name="add_sub" function="add"/>
 </Operation>
 <Operation name="ISUB">
  <Resource name="add_sub" function="sub"/>
 </Operation>
  ...
 </Operations>
</Config>
```

We can see on Listing 2.1 an example of a (partial) configuration file. In the "Operations" section, we see that operation IADD and ISUB are both implemented by resource named "add_sub". But the function is not the same. If we look now in the "Resources" section, we will see that a resource "add_sub" exists. There is a detailed description of the signals of this resource. We see that it has two inputs, the first labelled "in1" and the second labelled "in2" for instance. We see in its "Select" section that the selection signal is called "funSel". If this signal is '1', then an addition will be executed (whose delay is 1) and if it is '0', then the subtraction will be executed (whose delay is also 1, but it could have been different).

As we can see, this configuration file gives us a lot of flexibility. Finally, we can see on Listing 2.2 the component declaration of the "add_sub" resource.

Listing 2.2: The component that implements the "add_sub" resource (vhdl code).

```
component add_sub
port (
  clk    : in  std_logic;
  reset  : in  std_logic;
  enable : in  std_logic;
  funSel : in  std_logic_vector(0 downto 0);
  in1    : in  std_logic_vector(31 downto 0);
  in2    : in  std_logic_vector(31 downto 0);
  out1   : out std_logic_vector(31 downto 0)
);
end component;
```

Chapter 3

Abstraction from Java Byte-Code

It is important to have a good abstraction from JBC in order to simplify the next steps, such as schedule, code generation and optimisation. It also allows to have a representation adapted to the different algorithms that we will use. We have chosen to represent a Java program (composed of JBC instructions) by the mean of graphs. The first graph we will see is the Control Flow Graph (CFG). A CFG models flow of control in the program.

Next we will see that the nodes of this graph, known as basic block, are composed themselves of graphs. But let's first see how to build a CFG from a list of instruction.

3.1 Control Flow Graph

In order to build the CFG, we need first to detect the basic block boundaries [1]. Then, we will run the byte-code in an emulated way, in order to build the data flow graph of each basic-block.

The basic block is a sequence of instructions that always execute together (either they are all executed or none is executed). The choice of executing or not a basic block depends on control instruction. As we will see, the presence of such instruction will be responsible for the creation of the edges in the CFG construction algorithm. There is at most one control instruction per basic block and it is always the last instruction to be executed, in the original Java program. The result of this instruction will determine what is

[1] A basic block boundaries is determined by the first and the last instruction a basic block executes

the next basic block to be executed. In many assembler those instructions are the "branch" or the "jump" instruction.

JBC provides us with 4 kinds of branch instruction.

- GOTO instruction, which always jump to a known address
- IF instruction which may or may not jump to a known address depending on the result of the previous operation (if it does not jump, it simply follows the execution with the next instruction)
- JSR (Jump Sub-Routine) instruction that are not to be mixed with method call (in JBC, routine are different than method)
- SELECT instruction that is used to implement switch in the Java language (it jump to a known address depending on the argument, based on a table)

Our compiler only support the first two branch instruction kinds. But it would not be so hard to support the remaining ones.

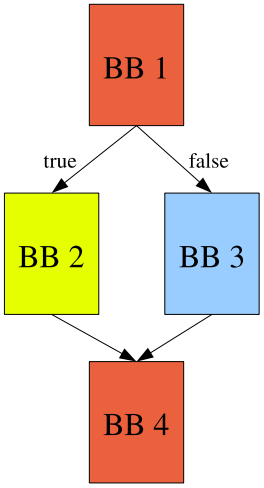


Figure 3.1: Example of a CFG : simple if then else statement. The basic block 2 (in yellow) is executed only if the result of the condition of the IF instruction is true. Else basic block 3 (in blue) is executed.

We can see the CFG corresponding to a simple *if then else* statement in Figure 3.1.

3.1.1 Basic block boundaries

To detect the basic bloc boundaries, we use a two-pass algorithm described in [5]. A first pass is done over the code to determine the leader instructions. An instruction is a leader if it is at the beginning of a basic block. In other words, the leaders set is composed of instructions whose execution occur right after a branch instruction.

We can see on Listing 3.1 the algorithm that find the leaders. As we can see, all instructions that immediately follow a branch instruction and all instructions that are the target of a branch are added to the leader set.

Once the leaders have been found, we create, for every leader, a node in the CFG. Then, we add the edges. Listing 3.2 gives the algorithm that adds correctly the edges and find the last instruction of each basic block. This algorithm first determines the last instruction of each basic block, then it creates the edges.

Listing 3.1: Algorithm that finds the leader instructions in the original JBC. An instruction is a leader if it is the target of a jump, the target of a branch or the instruction that follows a branch instruction.

```
Set leaders;
leaders.add(instructions.first);

for each instruction {

  if (instruction.isGOTO)
    leaders.add(next instruction);

  else if (instruction.isIF) {
    leaders.add(next instruction);
    leaders.add(instruction.branchTarget());
  }

}

return leaders;
```

Listing 3.2: Finding last instruction of the basic block and adding edges to the CFG.

```
for each leader instruction {
  BasicBlock bb = leader.getBasicBlock();

  // first, we find the last instruction
  We iterate from the leader instruction
  until we found a branch instruction;
  The instruction before the branch instruction
  is the end of the basic block;

  // next we create the edges
  if (leader.isGOTO)
    cfg.addEdge(bb, leader.target.getBasicBlock());

  else if (leader.isIF) {
    cfg.addEdge(bb, leader.target.getBasicBlock());
    cfg.addEdge(bb, leader.nextInstr.getBasicBlock());
  }

}
```

Here we have already found one advantage of JBC over other assembly codes ; the target of the branch instruction is perfectly known at compile time. There is no instruction whose jump address is stored in a register for instance. This is particularly useful since we are able to determine statically the basic block length and thus to schedule them prior to their execution. A big part of the problems commonly known to people that does dynamic translation of code are thus avoided.

Another problem, such as self-modifying code is also avoided, since it is not possible to alter JBC program during execution. Self-modifying code can be replaced with high-level concepts, such as dynamic method dispatching.

3.1.2 Data Flow Graph construction

Once we have identified the instructions that are part of a basic block, we are ready to construct a DFG (Data Flow Graph). This DFG encodes the data dependency between operations.

The construction of this DFG is done by emulating the stack and the local variable pool of the Java byte-code and by running (virtually) the byte-code.

This process is straightforward. Each time we execute (virtually) an instruction, we know which operands it takes by looking at the stack (in our

case the stack contains the operation that has produced the result, not the result itself). Then we create the corresponding operation, and we push it on the stack.

When this step is done, we end with the DFG and information about the stack and the local variables before and after the execution of the basic block. This information will then be used to determine the liveness value of the basic block and to do the verification process required by the JVM specification.

Let's look at an example. We can find in Listing 3.3 the Java source code of a simple method and its corresponding Java Byte-Code.

Listing 3.3: A simple Java method and its corresponding Java Byte-Code.

```
public void toto(int x, int y)
{
  int z = (x+y)*x;
  ...
}

// corresponding Java Byte-Code
iload_1     // load the first  parameter (x) on the stack
iload_2     // load the second parameter (y) on the stack
iadd        // addition
iload_1
imul        // multiplication
istore_3    // store the top of the stack in variable z
...
```

In Figure 3.2 we can see, step by step, the construction of the DFG and the state of the stack at each step. As described above, we push on the stack the operation that produce the result and not the result itself. The final DFG is shown on Figure 3.3. Normally each operation is unique (for simplicity reason we have represented the node with just the name of the instruction), since it is a different operation.

Remark now that the stack has disappeared ; it has been, in a certain way, encoded into the DFG.

3.2 Verification process

At the same time of the construction of the DFG, we can perform a part of the verification process described in section 4.9.2 of [4]. This step consists

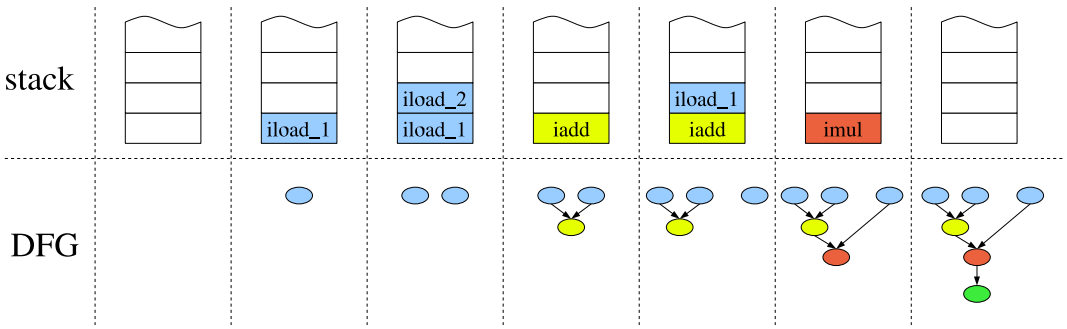


Figure 3.2: DFG construction example. The stack is represented at each step and below is the corresponding DFG. See Figure 3.3 for the details of the DFG.

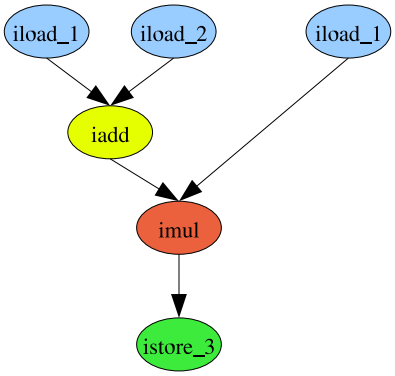


Figure 3.3: The DFG once its construction finished.

in checking that all operations get the right operand type (for instance an IADD operation requires two values on the stack of type INT).

By applying data-flow analysis technique to the state of the stack and local variable pool after and before execution (virtual execution done during DFG construction) of the basic block, we can verify that the Java program is correct.

Even if this step is not necessary for our compiler to do the synthesis, it is nearly free to do it at this moment. And as we will see in Section 7.3, it is easy to do it with the BCEL library.

Finally it is important to notice that this DFG is a DAG (directed acyclic graph). It will be useful for the application of some algorithms.

3.3 Liveness information

The liveness information tells us, for each basic block, what are the variables (local or stack variable) that are alive, at the beginning of the basic block and at the end. Those two sets are called *liveIn* and *liveOut* set. This liveness information is important, since it allows us to perform various optimisations, such as dead code elimination. Furthermore, it is particularly useful in our case, since it will allow us to correctly pass variables across basic block, as explained in Section 6.4.2.

To determine the *liveIn* and *liveOut* set, we first determine, for each basic block, its *def* set and its *use* set. The *def* set contains all the variable that the basic block defines while the *use* set contains the variable that the basic block read. It is clear now that the *use* set is what the basic block need as input and the *def* set what it produces.

But we have to take care about the fact that the successors of a basic block may need some variables that are not part of the predecessor *gen* set. The computation of the *liveIn* and *liveOut* set is a typical data-flow analysis problem. More details can be found in [5].

3.4 World dependency graph

The DFG is not sufficient to represent all the information contained in the original JBC. Some operations need to be executed before others. This is the case for memory operation for instance. This is why we need another graph ; the world dependency graph.

The world dependency graph represents the order between world operations. A world operation is an operation that access or modify the world (as described in Section 2.3). The operations that access the world have to be executed in between operations that modify it, in order to be conservative (for memory), relative to the initial instruction list of the JBC.

The creation of the world dependency graph is straightforward. We simply iterate through the operations (in the order defined by the initial instruction sequence), and as soon as we get an operation that modify the world, we add all the previous instructions till the one that modify the world as predecessors of the operation (this is not as difficult as it sounds).

We can see in Listing 3.4 the algorithm that constructs this graph. As for the DFG, the world dependency graph is a DAG.

Listing 3.4: World dependency graph construction. This graph encodes information usually known as RAW (Read After Write), WAR (Write After Read) and WAW (Write After Write) dependencies.

```
/* The operations are taken in the same order as their
   corresponding JBC instruction. */
for each operation of the basic block {

  add a node for the operation;

  if (operation.accessworld)
    add an edge from the last operation that modify the
    world to the current operation;

  if (operation.modifyworld)
    add an edge from every previous operation till the
    one that modify the world to the current operation;
}
```

3.5 Sequencing graph

Finally, in order to represent the information in a more convenient way, and to make the algorithms easier to applied, we "fusion" the DFG and the world dependency graph in a new graph called the sequencing graph. The construction of this graph is easy ; we insert all the nodes and their edges of one of the two graph in the other one, and we remove duplicated edges.

This graph defines an order that will have to be respected in order to guaranty correct behaviour of the synthesised Java program. This graph determines which level of parallelism we will be able to exploit. It is the fusion of both type of dependencies ; data dependencies and temporal dependencies (world dependencies).

Chapter 4

Resource sharing and binding

As we know, resources are implemented by LE. As the number of *logic elements* required to implement the resource can be huge, it is necessary to share some of those resources. Furthermore, some resources are unique in the system (like memory controller), thus resource sharing is necessary.

In our case, the FPGA we have used has about 4000 LE. When we know that a 32 bit multiplier that executes in one clock cycle take about 600 LE (15% of the available LE), it is clear that sharing is not an option, but a necessity.

Resource binding consists of assigning one physical resource for each operation. In the case where we have an infinity number of resources, this task become straightforward ; we simply assign a different resource to each operation. But in the presence of a limited amount of resources per kind of operation, this is not so simple. Furthermore, this task become more complex if the total number of resource of any type is limited.

We choose to do the resource binding prior to the scheduling. The reverse is also possible, but it would have do the scheduling task more complex. In the presence of *unbounded delay* operations, this choice let the schedule be easier to perform. The binding is done by constructing the *compatibility graph* for each type of resource and then by resolving *conflicts* between operations. Two resources are in *conflict* if they use the same resource at the same moment. To avoid such conflict, we serialise those operations in order to prevent that they execute in parallel.

We will first present the "classical" approach of resource binding and then present the way we have dealt with this problem in this project. As we will see, our solution is exactly equivalent to the "classical" approach.

4.1 Hypothesis and limitation

For time reason, we have made the hypothesis that we have an unlimited number of *logic elements*, but a limited number of resources for some operations, where other operations have an unlimited number of resources. In other words, we do not take into account the number of *logic elements* a resource use. Obviously this hypothesis is false, and it is an important weakness of our compiler. But actually this task would have taken too many time. As we provide a configuration file, that we can fill with information such as the number of available resources per operation, we can let a human player do the choice of which resources will be available in which quantities. But it is clear that to fulfil the requirement of automatic synthesis of JBC to hardware, this task has to be done inside the compiler without any intervention.

Some operations can be "implemented" in hardware by more than one resource type. For instance, if we take an addition operation, it could be implemented by an adder resource or by an ALU. But this problem, known as module selection, has not been taken into account. So, for each operation, we have only one resource type that can implement it. This problem is tightly coupled with the above problem of limited number of LE.

Another limitation is about pipelined resources. It would have been great to be able to deal with pipelined resources as such. We could have said that a given resource has a pipeline and that we can feed it with data every 2 clock cycles for instance. This would have allowed to reduce scheduling length and also the number of LE taken by the resource (because we transform the computation from the spatial domain to the temporal domain). But actually, we never activate again a resource if it has not finished its execution.

4.2 Compatibility graph

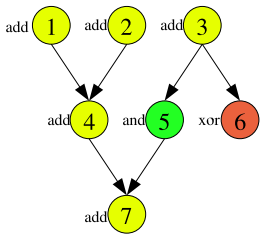
The "classical" approach is first to build the *strong compatibility graph*. Then we find the cliques in this *compatibility graph* and finally we do the binding and if needed some serialisations.

First let's define the notion of compatibility. Two operations are said to be *compatible* when they can be implemented by the same physical resource.

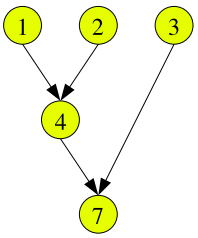
Now, we actually compute the *strong compatibility graph*. The notion of *strong compatibility* is the following : two operation are *strongly compatible* if they are serialised with respect to each other (i.e. when the corresponding nodes are the head and the tail of a directed path in the sequencing graph). It is clear that *strong compatibility* implies *compatibility*.

The *strong compatibility graph* is a undirected graph, its computation is the following : we visit the sequencing graph in DFS (Depth First Search) order and we create an undirected edge for each descendants of each node and the node itself. The sequencing graph on which this algorithm is applied is in fact the original sequencing graph compound only of operations that can be implemented by the same resource.

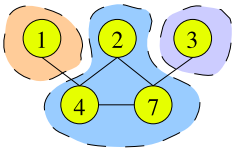
We can see on Figure 4.1 a sequencing graph, the filtered graph (we are interested only in ADD operations in this example) and its corresponding compatibility graph. The cliques in this last graph are represented.



(a) A sequencing graph.



(b) Filtered sequencing graph.



(c) Strong compatibility graph.

Figure 4.1: We can see the sequencing graph, its filtered graph corresponding to ADD operations and its corresponding compatibility graph (with three cliques).

One could wonder why we cannot just use the notion of compatibility ? The answer is simple ; as the scheduling has not yet been done, the only information that guarantee that two operations will not execute at the same time is the fact that their are serialised with respect to each other.

4.3 Cliques partition

In order to define the binding between operations and resources, we have to compute a maximum clique partition of the *strong compatibility graph* As we know that the *strong compatibility graph* is a comparability graph, there exist an algorithm that executes in polynomial time to partition this graph into cliques (this is the maximum clique cover problem).

We will not explain in details this algorithm since it is covered in [6]. Moreover we have not use this algorithm in our compiler, as will see in Section 4.5.

On Figure 4.1 we can see three cliques. The biggest is composed of operations $\{2, 4, 7\}$ and the two smallest of operation $\{1\}$ and operation $\{3\}$. Each operation of each clique can be assigned to the same physical resource, since we are sure that they will not execute at the same time (this is related to *strong compatibility* notion).

4.4 Operation serialisation

This step aims at resolving conflict in the case when we have less physical resources than we have cliques. In such a case we will have to serialise some *conflict operations* in order to bind them with an available resource.

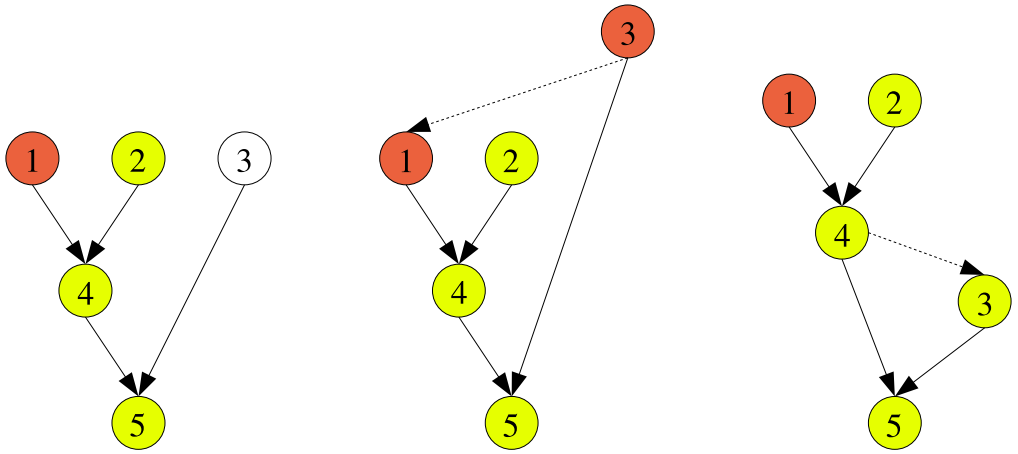
We will deal with one operation at a time. Intuitively, it seems that the order in which the operations are taken does not affect the final result. Unfortunately, we have not found enough time to verify this.

Let's look at an example. We can see on Figure 4.2 how it is possible to serialise operation number 3. We just show two possibilities, but others exist. Operations that are coloured with the same colour are bounded to the same resource.

The process of serialisation add new edges in the sequencing graph and this could increase the schedule's length. Thus we have to choose carefully where to serialise the operation (which edges to add). Furthermore it is clear that we do not want to create cycle when we add new edges. So we first have to identify the site where we can add edges safely.

4.4.1 Candidate places for serialisation

We have to insert our *conflict operation* in between two other consecutive operations that are in our path. We have also to take care to not insert cycle in our sequencing graph. Thus only a subset of operations in the path are candidates for the serialisation.



(a) Before serialisation.

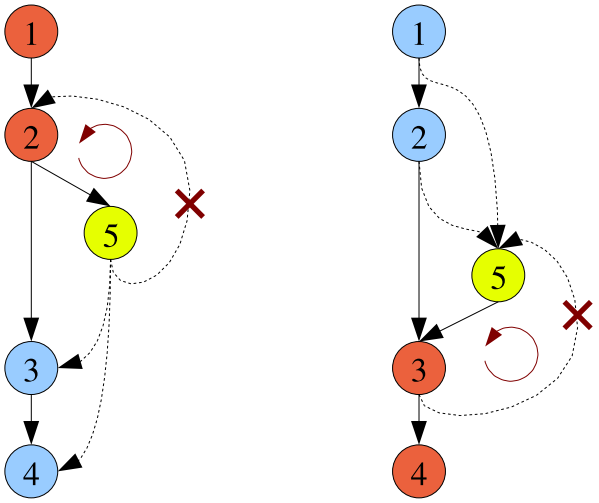
(b) A possible serialisation. Operation 3 is serialised with operation 1 and is thus bounded to the same resource as operation 1 (in red).

(c) Another serialisation. This time operation 3 is serialised with operations 4 and 5. It is bounded to the same resource as operations 2, 4 and 5 (in yellow).

Figure 4.2: Two possible serialisations of a *conflict operation*. Operation number 3 is the *conflict operation*. Those two serialisations lead to a different binding.

We can see on Figure 4.3 an example of two different cases. This figure represents a simplified sequencing graph. The operation number 5, in yellow, is the operation to serialise. In the first case only operations 3 and 4 can be chosen as successors of operations 5, while in the second example, only operations 1 and 2 can be chosen as predecessors of operation 5. Those examples illustrate the cycle that could be created if we choose the wrong operation as successor or predecessor. Note that the dashed edges are the ones that are possible to add (only 2 of the 3 does not create cycle).

Now that we have seen the problems that could occur, let's define more precisely the objective. The objective is to find the best place in which to put our *conflict operation* in the sequencing graph. To do this, we will, for each clique, extract the possible positions (thus avoiding creating cycle) and then compute a cost that will reflect the increase in term of clock cycle (or not).



(a) Ancestors of operation 5 create a cycle.

(b) Descendants of operation 5 create a cycle.

Figure 4.3: Candidates for serialisation. It is important to avoid the creation of cycle when doing serialisation.

Now that we have see intuitively where are the possible places to add edges, let's see that in a more formal way. We cannot insert the *conflict operation* before an operation in the transitive closure of the sequencing graph that is a predecessor of the *conflict operation*. And we cannot insert the *conflict operation* after an operation in the transitive closure of the sequencing graph that is a successor of the *conflict operation*. Those two conditions guaranty that the resulting sequencing graph will not contain any cycle.

4.4.2 The cost function

Having determined the candidates operations, we can now compute a cost function for the addition of the new edges, that will serialise the *conflict operation*. We define such a function in the next paragraph.

We denote by $c_{roots}(op)$ the critical path length from the roots to an operation *op*. $c_{leafs}(op)$ denotes the critical path length from an operation *op* to the leafs. The delay of an operation is denoted by $delay(op)$. Thus the length of a critical path from the roots to the leafs that goes through operation *op* is :

$$criticalPathLength(op) = c_{roots}(op) + delay(op) + c_{leafs}(op)$$

Special care must be taken with *unbounded delay* operations, since their delay is unknown. We choose to use an estimation of the delay, that is provided in the configuration file. This estimation can be based on the mean delay of the operation, or the best delay the operation can achieve. This last choice has been taken regarding memory operations. The first access to memory is very slow, because we are sure that the data is not in the cache. But for the next access, we make the bet that the data will be in the cache and thus the delay will be very low.

This cost function has to be minimised. We iterate through the possible place the operation can take in order to be serialised in the path and we compute the $criticalPathLength$ function. We choose definitely the position of the operation so that this function is minimised. When we have several possibilities, due to the fact that several positions give the same minimal critical path length, we choose the one at the most beginning of the path. See listing 4.1 for the algorithm. Furthermore, we iterate through the bounded cliques from the smallest to the largest one. Like this, we give priority to the smallest cliques (since the min_cost is updated only when the cost is greater). Thus, if we have the choice, we avoid the creation of too big cliques, which would means that some resources are more shared than others.

Listing 4.1: Serialisation algorithm. If there is more cliques than available resources, we have to serialise the remaining operations.

```
int min_cost = MAX_INT;
Position bestPosition;

Order the bounded cliques from the smallest to
the largest one;

for each bounded cliques {
  Path path = get the path formed by the clique;
  Retains only places that cannot create cycle;

  for each possible position in the path {
    insert the operation in the path;
    int cost = compute the cost with the cost function;
    if (min_cost > cost) {
      min_cost = cost;
      bestPosition = position;
    }
  }

}

serialise the operation at bestPosition;
```

4.5 Our approach

The problem of finding the cliques can be view as a problem of finding the longest path in the sequencing graph. Indeed two operations have an edge in the compatibility graph if they are on a path from one operation to the other in the sequencing graph. Thus a clique in the compatibility graph is simply a path in the sequencing graph. As we are looking for the maximum clique cover problem, we can equivalently solve the longest path problem in the sequencing graph. This is an advantage since we have no need to create a new graph and then to find the cliques.

As before, we proceed exactly with the same step, except that the computation of the clique is now reduced to the computation of the longest path in a directed graph. And we end up with a clique cover. This problem can be solved in polynomial time.

4.6 Improving resource usage

The resource usage could be optimised if we change the implementation of the multiplier, for instance. We could obtain theoretically a multiplier as small as wanted, but with an increase of clock cycles needed to do the computation (unless we give the possibility to use pipelined resource). This is a well known problem and we have not tackled it in this project. We just give the possibility to modify the implementation of the resources and to reflect those changes in the compiler by the mean of the configuration file, but this has to be done by hand.

Thus the problem of resource binding is related to resource sharing, which tends to share resource whenever possible. But we have to care about the interconnection logic in order to share resources, since interconnection logic could use a bigger number of logic elements that the operation we try to share. This is why we do not share too small resources, such as the one used by the AND operation by example. We will come back on this point in Section 6.2.

Chapter 5

Scheduling

The scheduling problem consists in assigning to each operation its start-time. The resulting schedule has to be consistent with the different order constraints that are imposed on the operations by the sequencing graph (which contains information about both DFG and world dependency graph). Furthermore, for a given operation scheduled at a given time, we have to be sure that all its operand values will be valid.

As we will see in Section 5.1.2, the operations are grouped inside partitions. In the presence of *unbounded delay* operations, the choice of *relative scheduling* proposed in [7] seems to be a good choice. Operations that are in the same partition are known to all depend on the same set of *unbounded delay* operations to start their execution. Thus we can schedule those partitions with a classic approach. This approach consist in determining the start time of each operations, relative to the *anchors* of the partition. This is *relative scheduling*.

As resource binding has already been done, the scheduling problem is simply reduced to find an optimal schedule, given the sequencing graph resulting of the binding step. We do not care anymore about operation conflicts, since resource binding step has serialised operations according to the *strong compatibility* notion.

5.1 Creating the partitions

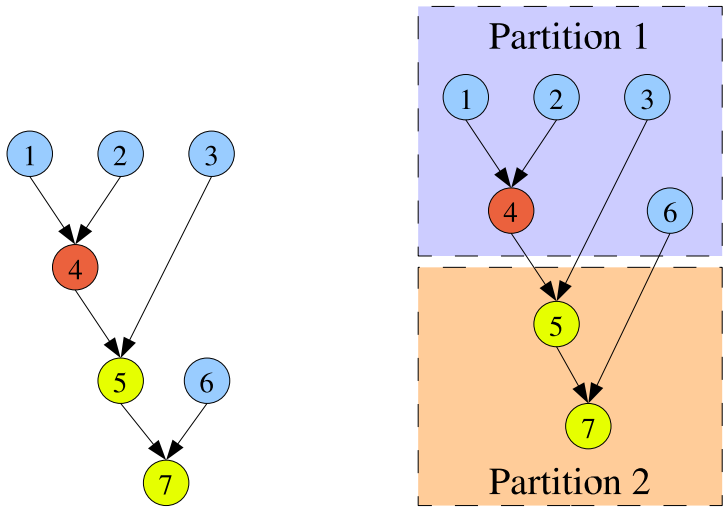
As introduced above, we define a partition by a set of operations which have to wait on the same set of *unbounded delay* operation to begin their execution. If a partition contains *unbounded delay* operations, they cannot have successors in the same partition (otherwise it would violate the definition of a partition).

5.1.1 Anchor set

In order to create the different partitions for a given basic block, we have first to compute, for each operation of the basic block, their anchor set.

The anchor set of an operation consist of all the operations that have an unbounded delay that affect directly its start-time. The anchor set of an operation acts as a kind of synchroniser. Note that the fact that all operations in an anchor set of a given operation have finished their execution is not a sufficient condition to allow the start of the given operation, but a necessary one.

5.1.2 Partitioning the graph



(a) A sequencing graph. Operation 4 (in red) has an unbounded delay.

(b) The resulting partition.s

Figure 5.1: This sequencing graph creates two partitions, since operation 4 has an unbounded delay. Operations 5 and 7 are in the same partition since they both depend on the same unbounded operation. Each partition can be scheduled statically.

From the anchor sets computed at the previous step, we can now extract the partitions. A partition is simply a set of operations that have all the same anchors set. In other words, every operation in the same partition depends on the same set of *unbounded delay* operations to begin their execution.

Creating the partition from the anchors set of each operation is simple : we just group the operations whose anchors set is identical.

We can find in Figure 5.1 an example of the partitioning of a sequencing graph. Operation number 4 has an unbounded delay (in red), thus the anchor set of operations 5 and 7 contains only operation 4. We end with two partitions, that can be scheduled at compile time (see Section 5.3).

5.2 Scheduling a basic block

The scheduling of a basic block involves the scheduling of all its partitions. It can be done without any particular order, the schedule of a partition will always be relative to the predecessors partitions (and the end time of a partition is unknown, since it is composed of *unbounded delay* operations).

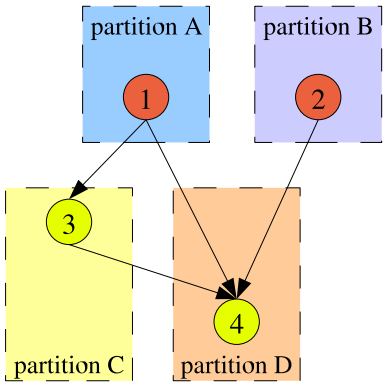


Figure 5.2: Example of dependency between operations belonging to different partitions. Operation 4, which is in partition D, also depends on operation 3, which is in partition C. Thus the start-time of operation 4 has to be later than operation 3 start-time.

But it is important to notice that there can be dependencies between operations of different partitions. This is illustrated in Figure 5.2. Operations 1 and 2 (in red) have an unknown delay and are the anchor of partition C and partition D. Partition C anchor is operation 1 and partition D anchors are operations 1 and 2. We can see that operation 3 depends on operation 1 and thus belongs to partition C and that operation 4 depends on operations

1, 2 and 3 and is in partition D. Despite the fact that operations 3 and 4 (in yellow) are in different partitions, a dependency between them exists.

If we look at this example, it is clear that partition D will always start its execution at the same time or later than partition C (as it has to wait on one more *unbounded delay* operation than partition C). In order to guarantee that the dependency between operations 3 and 4 will be respected, operation 4 start time (relative to the start of partition D) has to be greater than operation 3 start time (relative to the start of partition C).

We will see in the next sections how this case is handle.

5.3 Scheduling a partition

We schedule the operations according to the *ASAP* (As Soon As Possible) algorithm. The complexity of this algorithm is linear. This algorithm gives us the optimal schedule, in the sense of minimal schedule length, for a given resource binding.

Listing 5.1: ASAP algorithm. The operations are scheduled as soon as possible.

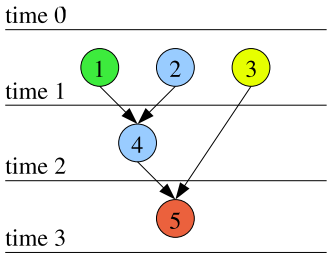
```
List candidates;
Set done;

while (! candidates.isEmpty) {
  Operation op = candidates.first;
  int startTime = computeStartTime(op);
  scheduleOperation(op, startTime);

  candidates.remove(op);
  done.add(op);

  for each successor of op {
    if done.containsAll(succ.predecessors) then
      candidates.add(succ);
  }
}
```

We can see on Listing 5.1 the ASAP algorithm. The predecessor method refers here to the predecessors of an operation in the same partition. If the predecessor comes from another partition, no check are done at this level. This is the computeStartTime method that will actually to the necessary things ; schedules the partition of any predecessors operation whose partition has not yet been scheduled.

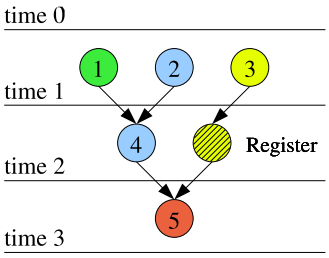
Let's see more in details what computeStartTime and scheduleOperation methods do. But before let's look more in details about the problem of validity of result.

5.3.1 Ensure validity of operands

As mentioned, we have to take care about the fact that operands of each operation are valid at the time the operation is scheduled. To guaranty this, we have to insert, sometimes, s special operation : a register operation. This operation is implemented, as its name suggest, by a register. Once scheduled, the register will hold the value until the operation that need it has begun. Figure 5.3 shows the problem and the register operation added to resolve it.



(a) Problem here : result of operation 3 has to wait 2 cycles. The result could not be valid at time 3.



(b) The problem is resolved by the addition of a "register" operation.

Figure 5.3: We must guaranty that each operation gets a valid input.

However, as seen in Section 2.4.1, if the operations respect their contract, the result of the operation is valid until the next execution of the operation. The register added in figure 5.3 is thus unnecessary, because operation 3 result will be valid at time 3. But consider the case where operations 3 and 4 would have used the same resource. In this case, the result of operation 3 would not have been anymore valid at time 3, and the register would have become necessary.

As we will see, the scheduleOperation method take care of this (Section 5.3.3).

5.3.2 The computeStartTime method

The computeStartTime method computes the time at which the current operation can be executed. Listing 5.2 gives all the details about this method.

We compute the maximum start time by iterating over every predecessors of the operation (here we take not only the predecessors of the same partition, but also those of the others). If a predecessor has an unbounded delay, we have nothing to do, since it will be part of a partition that we have to wait on, to start the execution. If the partition of the predecessor is not the same, we check if it is an ancestor of the actual partition. If this is not the case, we have to take care of this case (it is the same as explain in Figure 5.2. We thus schedule the partition that has not yet been scheduled and then we do as if the predecessor where in the same partition. Finally we retrieve the start time of the predecessor (based on a table) and we check if this start time plus the delay of the predecessor is greater than max. If so, we update max value. Once every predecessors have been "visited", we return max value, which the start time of the operation.

Listing 5.2: The computeStartTime method. This method is responsible for the computation of the start-time of an operation.

```
int max = 0;
for each predecessors of op {
  if pred.hasUnboundedDelay
    continue;
  if (pred.partition != op.partition) {
    if (pred.partition is an ancestor of op.partition)
      continue;

    if (pred.partition is not yet scheduled)
      schedule(pred.partition);
  }
  int pred_start = pred.getStartTime();
  if (pred_start+pred.delay > max)
    max = pred_start;
}
return max;
```

As defined by ASAP algorithm, this method return the littlest possible start time of an operation, based on the start time of its predecessors.

5.3.3 The scheduleOperation method

This method actually schedules a given operation at a given time. This method does two things, first it stores in a table the start time of the operation and secondly, it checks if some register need to be added to operands of the operation.

Listing 5.3: The scheduleOperation method. This method schedule a given operation at a given time.

```
scheduleTable.add(operation);

for each predecessors of operation {
  if (pred.isConstant || pred.isRegister)
    continue;

  if (result of pred is not valid at this time)
    put the result of pred in a register;
}
```

We can find in Listing 5.3 the pseudo-code of this method. Note that when we put the result of one of the predecessor in a register, we modify the DFG. The inserted registers are only used to store the result of a specific operation at a given time. Once a value is stored in those registers, it will never get cleared of overridden in the same execution of the basic block.

Last but not least, the validity check is not as easy as it could appear. We have to maintain a table for each partition, that indicates, for each operation, the time at which the result become invalidated (by another operation who use the same resource). Each time we schedule a new operation, we have to update those tables.

5.4 Improvements

Some improvements could be done to the scheduling. ASAP algorithm gives us an optimal schedule with a minimal resulting schedule length. But other algorithm, such as ALAP (As Late As Possible) gives also minimal schedule length. ALAP algorithm schedules operations as late as possible in the time. As seen in section 5.3.1, we have to place registers when appropriate in order to ensure that operand results are still valid when executing the operation that need them. By choosing a mixture of ASAP and ALAP, we could

reduce the number of inserted registers, because we could, in certain cases, delay the start-time of some operations, in order to keep valid a result for another operation longer in the time.

An interesting point to raise is the fact that the smaller the available resource number is, the bigger the resources are shared. And an increase in the number of shared resources means an increase in the number of registers to add (the results of operations are likely to become less valid in the time, since the resources are more often used). Thus it is important to have a good schedule algorithm that does not insert too many registers, by moving whenever possible the operations in the time, without modifying the schedule length. Otherwise all the benefit of reusing the same resource many times could become a disaster if the scheduler inserts too many registers.

Chapter 6

Mapping to hardware

6.1 Overview

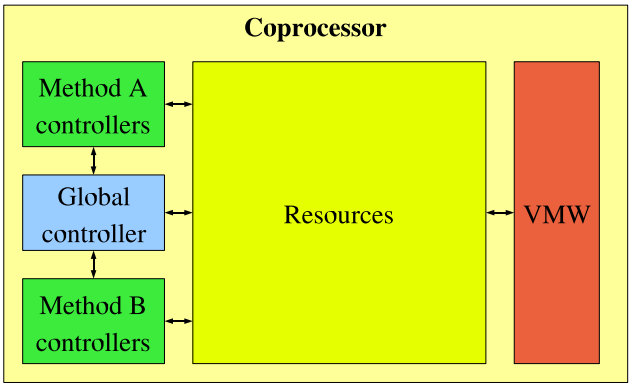


Figure 6.1: Our hardware model. As we can see, the different controllers of the different methods do not directly access the VMW interface. They deal only with resources. Thus a special resource is responsible for the communication with the VMW interface.

The hardware model we have chosen is to have the resources at the heart of the system. All the different controllers (of the different methods) control the same set of resources. We can see this on figure 6.1. A global controller is responsible for managing the start of the execution of the coprocessor and its end. But it does not control the method controllers. It only waits on some signals from them in order to know when the execution is finished. As we can see, every action that comes or goes to the outside of the coprocessor goes through the VMW. Except in some rare cases, such as start and finish signals, the different controllers does not access directly the VMW interface.

They access it indirectly by activating the resource responsible for communication with the VMW interface. A special resource is thus responsible for the communication with VMW. This resource performs all memory operations (read and write), but not only. It also performs operations to read the parameters of the method and operations that return the value once the method has been executed ("method" refers here to the main method that has been implemented in the coprocessor).

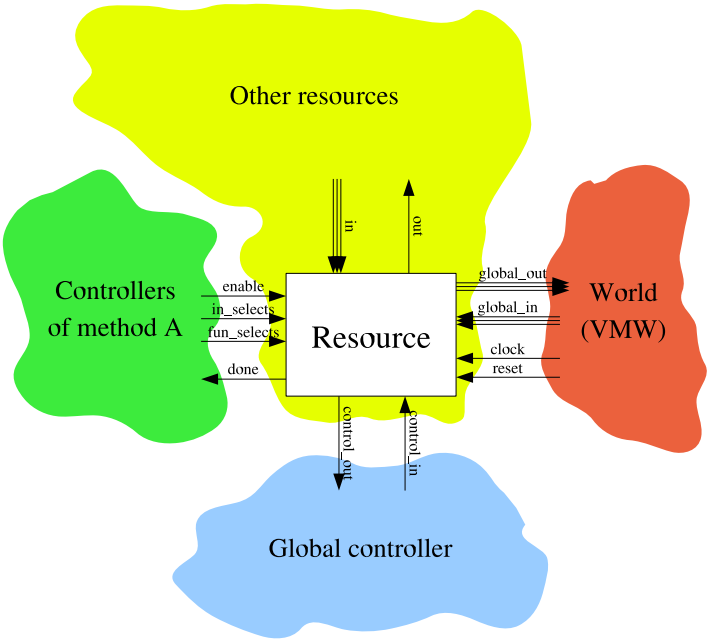


Figure 6.2: Resources are the heart of the system.

We see another view of the system in Figure 6.2. we see on this figure the resource that is responsible for the interaction with the VMW interface (the world). The global controller use this resource to access the VMW too (which simply redirect some signals of VMW).

6.2 Resource inter-connection

As seen in the previous section, the resources are at the heart of the system. We can see in Figure 6.3 how the resources are inter-connected and how the different controllers are connected to the resources.

By looking at this picture we can better understand why the inter-connection logic has a cost. For every input of each resource, we have multiplexer (im-

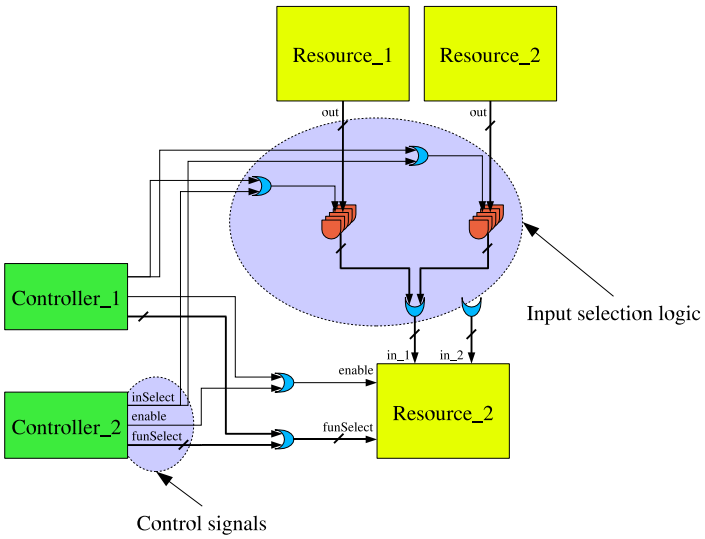


Figure 6.3: Resources inter-connection. A resource can be used by more than one controller. Furthermore the inputs of the resource can come from more than one resource.

plemented by AND and OR gates here). As those inputs signal are generally 32 bit width, it is as many AND and OR gates !

Thus the cost of inter-connection between resources would have to be taken into account when doing resource binding, since sometimes it is preferable to use distinct resources instead of sharing only one, even if their do not executes in parallel. Unfortunately we have not had time to take this into account.

6.3 Basic block

A basic block is composed of several controllers that generate the interconnection signals to drive the data among the different operations. As seen in Section 3.3, the liveness information is used to determine which variables (local or stack variable) are used by the basic block and which are produced. We have seen that with the liveness information we were able, for each basic block, to say which variable it has to receive and which it has to transmit. Thus, we can see a basic block as in Figure 6.4. It is composed of a set of registers, which are used to store the variables that are in the *liveIn* and *liveOut* set. The first operations of the basic block will be the LOAD operations that will load the value from the registers in order to use them in computation (if

needed), and the last will be STORE operations that will store some results in the register (again only when needed).

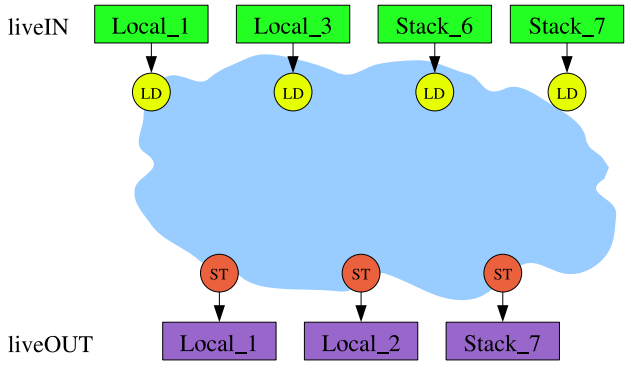


Figure 6.4: A basic block with its *liveIn* and *liveOut* set of registers. The rectangles represent registers for variables. The basic block loads its values from the variables, perform some operations and then store back the results in the variables.

As we will see in Section 6.4.2, the exchange of data between basic blocks will be done by the use of those shared registers, that represent the local and the stack variables.

6.4 Controller generation

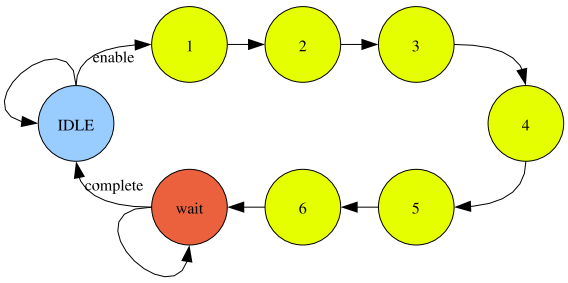


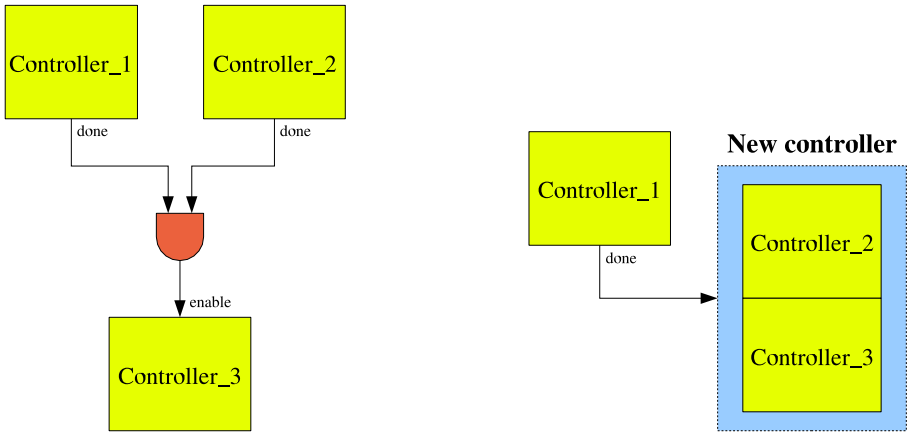
Figure 6.5: A finite state machine representing a schedule. There is an IDLE state (in blue) which wait for the start of the state machine, then the yellow states are used to activate the different operations as defined by the schedule. And finally a WAIT state (in red), which is optional, wait on *unbounded delay* operations (if any) to finish the execution.

We have chosen to implement the controller as a finite state machine. Another possibility we have tried, was to implement the controller by the

mean of a ROM (micro-code). But we have abandoned this idea, since the FPGA limits us in the number of ROM modules available, and we have had more controllers than available ROM modules.

There is a direct mapping between the schedules of the different partition and the finite state machine. We can see such a finite state machine on Figure 6.5. Each of the state represents a set of operation to activate. The last state (optional), is the "wait" state in which the state machine wait until every operation with an unbounded delay has finished their execution. Once the *complete* signal is received (from the *unbounded delay* operation), the state machine goes into the IDLE state. When doing so, a signal is emitted to inform the rest of the system (the other state machine that are waiting on it) that this state machine has finished its execution. Another state machine will probably then begin its execution.

6.4.1 State machine fusion



(a) Controller 3 has to wait on completion for Controller 1 and Controller 2. This generates 3 state machines.

(b) By fusioning the state machine of Controller 2 and Controller 3, we can reduce the number of state machines.

Figure 6.6: Fusion of two state machines.

One of the main problem with this simple approach is the number of state machine generated. There is as many state machines as partitions. This number is huge, since there is one partition for each operation with an

unbounded delay (such as memory operations).

So in order to resolve this issue, we have tried to reduce the number of state machines by fusioning whenever possible the state machines. Such a situation can be seen on Figure 6.6.

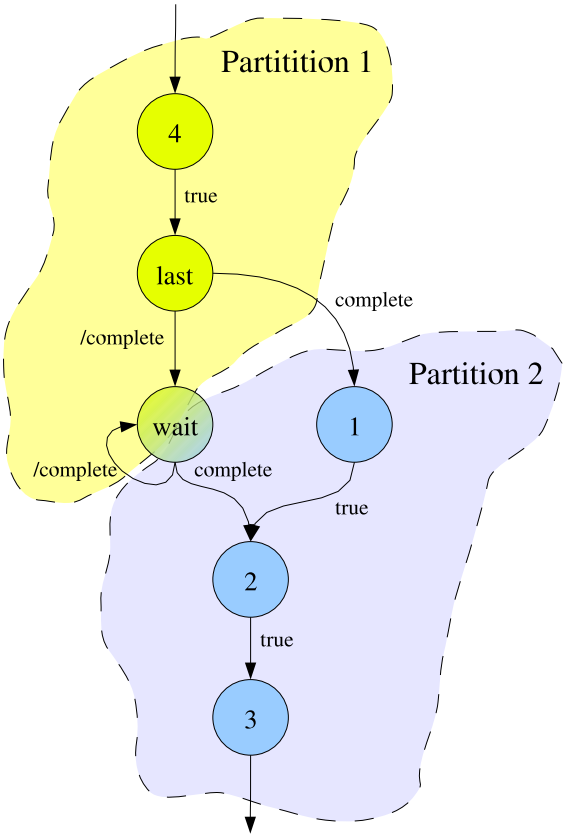


Figure 6.7: Details of the "fusioned" state machine. The "last" state of partition 1 actually performs the last operations of partition 1. If every *unbounded delay* operations have finished their execution, the "complete" signal is thus received and the state machine goes into the first state of partition 2. If it is not the case, it goes into the wait state. Once the "complete" signal is received, we directly emit the activate signals for state 1 of partition 2 and we jump to state 2 of partition 2. Like this, we can react faster.

We can see in Figure 6.7 the states of a "fusioned" state machine. In order to be more reactive, the "wait" state of the first state machine contains the activate signals of the first state of the second state machine. Those signals are enabled when the transition condition is active. Thus we end with a Mealy machine instead of a Moore one. Note that the "wait" state

is optional (it is possible that we do not need to go in this step), since the complete signals could already have come at the state named “last”.

6.4.2 Flow control

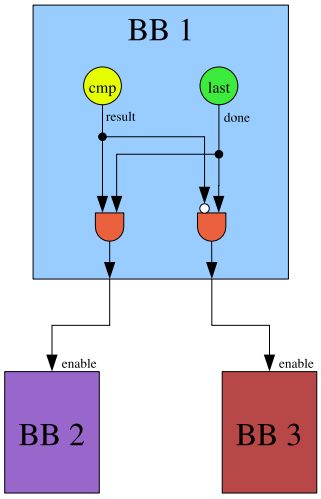


Figure 6.8: Flow control. The result of the CMP operation determines what is the next basic block to execute. If it is ’1’, basic block 2 is executed, else basic block 3 is executed.

The control of the flow, which is described by the CFG, is done without any general controller. As seen, the whole structure is a kind of network of controller which communicate by the mean of “finish” and “enable” signals. In order to do flow control, the last controller of each basic block (which contains the control operation) is connected to the first controller of the successor basic blocks, by an enable signal. There is one enable signal per successor. If we take the simple case of a CMP operation that provides the result of the control operation, only two successors are present. We can see on Figure 6.8 this case. It does not require a lot of logic (two AND gates and an INVERSER) and it works well.

One of the big advantage of having stored all the variables (local or stack) in shared register is the fact that it is not necessary to exchange explicitly (by the mean of signals) data across basic block. The main disadvantage is the fact that the registers are very connected to other resources.

6.5 Method calls

An interesting feature that we have implemented is method calls. It allows the coprocessor main method to call other methods that are in the same coprocessor.

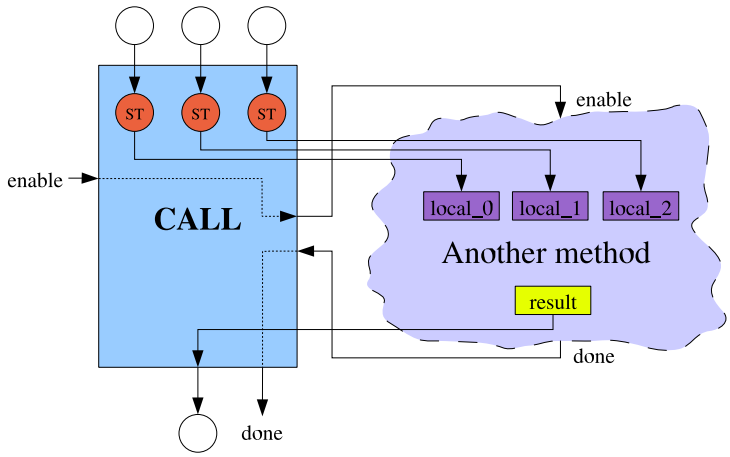
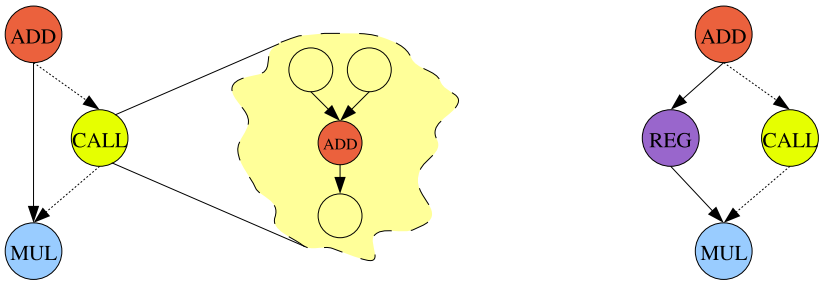


Figure 6.9: Call to another method that is inside the coprocessor. From the caller method, we store directly into the variable registers of the called method the parameters. The result of the called method (if any) is directly taken as the output of the call operation. We do not need to pass it through register since the result of the called method will always come from the same operation.

We can find in Figure 6.9 how method call works. It consists of simply copying the parameters of the method to call in its register representing the local variables (in Java and in our hardware model, the parameters are passed by the mean of local variable). Those registers are those of the method to call, and have nothing to do with the ones of the caller. Once the parameters have been stored in those registers, the start of the method happens immediately (the enable signal of the call operation is directly connected on the enable signal of the first controller of the called method). The return value, if any, will always come from the last operation that the called method "execute". The output value of the CALL operation is thus reduced to the output of the last operation of the called method.

As the resources are shared across the methods, the CALL operation always executes alone ; no other operation of the caller method are executed while we wait on the CALL operation to finish. But this is not sufficient, we also have to be sure that all the result of the previous operations are correctly stored in registers (remember that by contract, a resource store its

result until its next execution).



(a) Problem with the ADD resource ; it is used inside the called method too. The result of the ADD operation is not anymore valid after execution of the called method.

(b) Problem resolved by adding a register.

Figure 6.10: Problem of result validity when a method call occurs. We have to take care about the fact that the called method could invalidate the result of some operations in the caller method.

Consider the case of Figure 6.10. In this case we need to add a register in order to preserve the validity of the result of operation ADD, since it is also use in the called method. This problem is simply solved by taking care of this in the scheduling step, as described in Section 5.3.1. We simply use a trick that consists in doing as if the CALL operation was bounded to every possible resources. In other term, we do as if the called method could use all the available resources.

It is clear that this supposition is too strong and we could easily improve it by restricting to only the resources the called method really used (this has not been done because the time was missing). As we will see in the IDEA coprocessor example, this could greatly improve the performance.

Some work remains to be done to support more than a simple call. One could implement recursive calls for instance. But this is left for future work.

Chapter 7

Compiler details

The task of building a compiler from scratch is not an easy one. The compiler is actually more a proof-of-concept than a production compiler. Our compiler is not a pure model of software beauty and cleanness, but is actually working. Many of the classes that composed the compiler have been written nearly from scratch more than once. In fact the construction of this compiler has be done in an iterative way, by adding new functions as problems occurred.

We have chosen to develop this compiler in Java. Even if some people argues that Java is slow, it has the advantage of providing good abstraction such as the java.util packages who offer classes to manipulate collections. By using those high-level classes, such as Hashtable or HashSet, the resulting compiler is not too slow and we have the advantage of a fast development time. Furthermore the paradigm of OOP (Object-Oriented Programming) ease the development. To cut a long story short, is not this project about the synthesis of Java programs ?

Even if nearly all method are well commented (by the help of Javadoc), the different interfaces need to be cleaned. As we will see, the overall structure is in place, but some reorganisation needs to be done.

Figure 7.1 explains the required steps that produce hardware from JBC.

7.1 General structure

The compiler has been sub-divided into 4 packages. The first one is a graph package, that provides many convenient methods to manipulate graphs.

The second package is an hardware generator package. It allows actually the generation of VHDL code.

The third one is the main package that contains the compiler. The last one is a sub package of the main package that contains all the operation

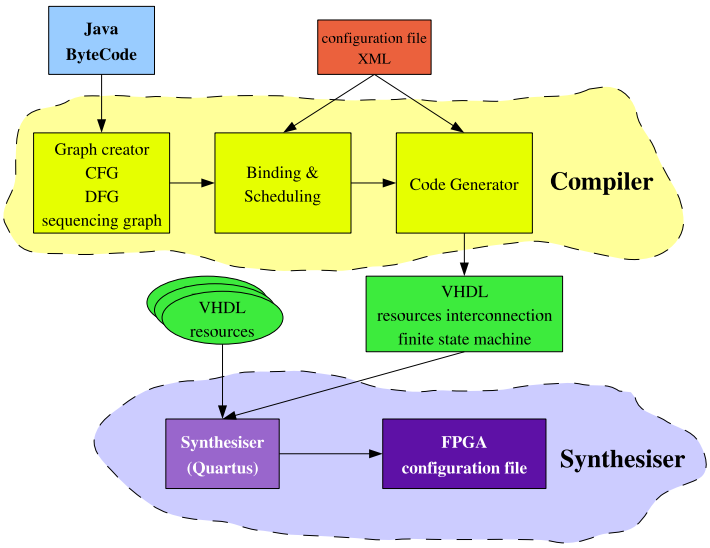


Figure 7.1: This figure explains the path from JBC to hardware. First, the compiler creates the different graphs from JBC. Then the binding and the scheduling is done, using information about the resources contains in the configuration file (such as the number of available resources and their delay). Next, the compiler generates a VHDL file which describes the interconnection between the resources and the finite state machines resulting from the scheduling step. At this stage we need once again the configuration file in order to get the details about the resources (such as the name of the input signals for instance). Next we feed the synthesis tool (Quartus in our case) with the "static" implementation of the resources and the generated VHDL file. It finally produces a configuration file that will programs our FPGA.

classes. Each operation has a corresponding class. Sadly, with the time, this last sub-package as been fill up with other classes, that would have to be moved elsewhere now...

7.1.1 Graph package

As we are working with many graphs, we have implemented a package for graphs. It provides many methods to manipulate graphs and could be reused in another project.

Here is a brief overview of the methods this package provides :

- visitor methods that allow to navigate through a graph (pre/post order, reverse DFS and so on...)
- transitive closure and transitive reduction of a graph
- finding the longest path in a graph (using dynamic programming)
- printing of the graph in a dot format
- many other methods that allow manipulation of a graph

As it is the case for the whole compiler, this package needs some cleanup in order to be usable for another project. But it would not require a lot of time to reach the quality criterion of a good Java package.

7.1.2 Hardware generator package

This package was first designed to provide an hardware abstraction that could be used to generate code in any languages. Sadly, with the addition of new functions in the compiler, this package has evolved into a VHDL generator only. But it would not be difficult to redesign it to be independent of the language. It would also be possible to completely isolate it from the compiler, in order for it to be used in other projects.

It contains some base primitive such as AND or OR gates, and some classes to support high-level hardware pieces such as finite state machine.

It could be greatly improved by providing a "view" of the hardware for instance, as many synthesis tools provide. This would help for the debugging of the compiler. The generated VHDL file is often huge (thousands of lines), thus it could definitely be a good idea.

It would also be interesting to have a look at *BOOM* (Berkeley Object-Oriented Module-Generators)[1]. This project is a generator framework written in Java that can be used as a basis for programming reconfigurable devices. It has many features such as simulation and is able to optimize the design relative to the technology used. More information about this project can be found in [8].

[1] http://brass.cs.berkeley.edu/BOOM/index.html

7.2 Compiler flow

The first thing the compiler does it to create the CFG of the method to synthesise. This involves the parsing of the method's code. This is done by using the BCEL library.

As described earlier, next we create the basic block and we build their DFG. Once it is done, we launch the verification process on the CFG in order to verify that the Java program is valid. Next we run the liveness analyser to compute the liveness information.

Once done, we do, for each basic block, the following :

1. we remove dead operations in the DFG
2. we compute the world dependency graph
3. we merge the world dependency graph and the DFG into the sequencing graph
4. we do the binding

Next we remove any empty basic block, resulting from the dead code elimination.

And we keep on by doing, on each basic block :

1. the creation of the anchor sets
2. the creation of the partitions
3. and the scheduling of the partitions

Finally a third pass is done over every basic blocks in order to assign for each input of each resources the resources it will take the value from. This pass uses only the information contained in the DFG.

Next, we do the same thing for every method that have been called. Once done, we begin the generation of the code. This step involve first the declaration of every signal that the resources used and the generation of the resource instances. Then we generate every controller, and finally we generate the resource interconnection logic.

7.3 DFG construction

BCEL provides to us a Visitor that we use to create the DFG. A Visitor is a design pattern. It encapsulate an operation that we want to perform on the element of a data structure. In our example, the operation we want to perform if the creation of a DFG. As each instruction (which has been parsed) has a method called "accept" that takes, as parameters a visitor, it will call the corresponding method in the visitor. Let's look at a simple example in Listing 7.1.

Listing 7.1: A piece of the visitor that creates the DFG. The visitIADD method will be called by the instruction when we will call its accept method.

```
...

public void visitIADD(IADD obj)
{
  // create the IADD operation
  Operation op = new IADDop();

  // add the operation to the DFG
  dfg.addNewNode(op);

  // pop from the stack the two operands
  Operation operand2 = pop();
  Operation operand1 = pop();

  // add two edges in the DFG
  dfg.addNewEdge(operand1, op);
  dfg.addNewEdge(operand2, op);

  // push the IADD operation on the stack
  push(op);
}
...
```

The visitIADD method is called by the accept method of the IADD instruction. Here is in Listing 7.2 the code of the accept method of the IADD instruction (taken from BCEL source and filled with comments). In order to create the DFG, we simply iterate over the instruction list and we call the accept method on it with the appropriate visitor (in fact we call 3 visitors, one that executes the instruction, one that verifies the operands and our visitor that build the DFG, the two others being offered by BCEL).

Listing 7.2: The accept method of the IADD instruction (taken from BCEL sources). This method calls the method implemented in the visitor ; visitIADD.

```
...

public void accept(Visitor v)
{
  /* This method call all the appropriate methods
  of the Visitor interface. */

  // it is a typed instruction
  v.visitTypedInstruction(this);

  // it produce a result on the stack
  v.visitStackProducer(this);
  // and it takes values from the stack
  v.visitStackConsumer(this);

  // it is an arithmetic instruction
  v.visitArithmeticInstruction(this);

  // and finally it is the IADD instruction
  v.visitIADD(this);
}
...
```

7.4 Final note

Despite the fact that the compiler needs to be improved on several fronts, it is already very flexible. It is easy to replace the actual operation scheduler for instance or to modify the way finite state machine are generated (by generated micro-code for example).

It is also ready for the addition of information about the resources. It would be interesting to have for each resource, the number of interconnection, or the number of time they are shared. Such information would improve to choose the appropriate algorithm in order to reduce resource usage and thus area on the FPGA.

Actually our compiler supports only a part of all existing ByteCode instructions. It supports only instructions that act on integer (it lakes support for float, double and long). Even if it does not support all arithmetic instructions, it is very easy to implement new instructions (we just have to create a VHDL file corresponding to the operation, to add its information in the configuration file and to implement the visitor "accept" method in the class

responsible for the creation of the DFG.

Other kind of instructions, such as method invocation, are not implemented. But it would also be easy to support them (the native library is ready to support callback from the coprocessor, as described in [1]). Instructions that access field of a Java class are also not supported, but they are similar to method invocation instructions.

Another important missing feature is the support for exception and for synchronisation primitive. Exception support could be added at the CFG level, but the synchronisation feature would be more complex to implement.

Chapter 8

A real case study : the IDEA coprocessor

Now that we have seen the path from JBC to VHDL, let's take a look at a real example ; the IDEA coprocessor. IDEA[1], is an cryptography algorithm. One particularity of this algorithm is the fact that it does not need any table (contrary to many cryptography algorithm), as we will see in the next section. This is particularly useful in our case, since our compiler does not support access to class fields (which are generally used to store the table).

8.1 Description of the algorithm

This algorithm is applied on blocks of 64 bits using a key of 128 bits. It consists of the application of 8 rounds and an output transformation. The following operations are used :

- bitwise exclusive or
- addition modulo 216
- multiplication modulo 216+1, where zero is interpreted as 216

We can see on Figure 8.1 one of the encryption round.

Both encryption and decryption are done using the same function (this function will be refered as the encryption function). The unique change between encryption and decryption that we have to do, is to modify the key. We will see in the next section the encryption function, as it is what we have synthesised in the FPGA.

[1]International Data Encryption Algorithm

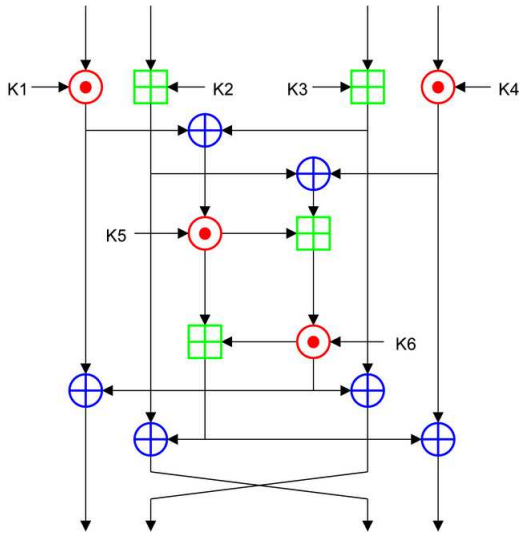


Figure 8.1: An IDEA encryption round. XOR operation is represented by a blue $\oplus$, addition by a green + in a box and multiplication in a red $\odot$.

8.2 Java source

We can find in Listing 8.1 the Java source code of IDEA encryption algorithm we have used. We can see the code of method *bytesToWord* on Listing 8.2, *wordToBytes* on listing 8.3 and *mul* on Listing 8.4. Those 3 methods are called within the encryption method.

If we take look at the IDEA Java source code, we will see that we do an external loop. In fact we do not just compute 64 bits, but we compute $64bits * blockNum$. We are doing this because it would be too costly (in term of overhead) to call many times the coprocessor from outside.

Listing 8.1: The IDEA encryption algorithm.

```
private static void ideaFunc(
  int[]   workingKey,
  byte[]  in,
  int     inOff,
  byte[]  out,
  int     outOff,
  int     blockNum)
{

  for (int block = 0; block<blockNum; block++) {
    int     x0, x1, x2, x3, t0, t1;
    int     keyOff = 0;
```

```
        x0 = bytesToWord(in, inOff);
        x1 = bytesToWord(in, inOff + 2);
        x2 = bytesToWord(in, inOff + 4);
        x3 = bytesToWord(in, inOff + 6);

        for (int round = 0; round < 8; round++)
        {
          x0 = mul(x0, workingKey[keyOff]);
          x1 += workingKey[keyOff+1];
          x1 &= MASK;
          x2 += workingKey[keyOff+2];
          x2 &= MASK;
          x3 = mul(x3, workingKey[keyOff+3]);

          t0 = x1;
          t1 = x2;
          x2 ^= x0;
          x1 ^= x3;

          x2 = mul(x2, workingKey[keyOff+4]);
          x1 += x2;
          x1 &= MASK;

          x1 = mul(x1, workingKey[keyOff+5]);
          x2 += x1;
          x2 &= MASK;

          x0 ^= x1;
          x3 ^= x2;
          x1 ^= t1;
          x2 ^= t0;

          keyOff += 6;
        }

        wordToBytes(mul(x0, workingKey[keyOff]),out, outOff);
        wordToBytes(x2 + workingKey[keyOff+1], out, outOff+2);
        wordToBytes(x1 + workingKey[keyOff+2], out, outOff+4);
        wordToBytes(mul(x3, workingKey[keyOff+3]), out, outOff+6);
      }

      outOff += 8;
      inOff  += 8;

    }
```

Listing 8.2: The *bytesToWord* method.

```
private static int bytesToWord(
  byte[] in,
  int inOff)
{
  return ((in[inOff] << 8) & 0xff00) +
          (in[inOff + 1] & 0xff);
}
```

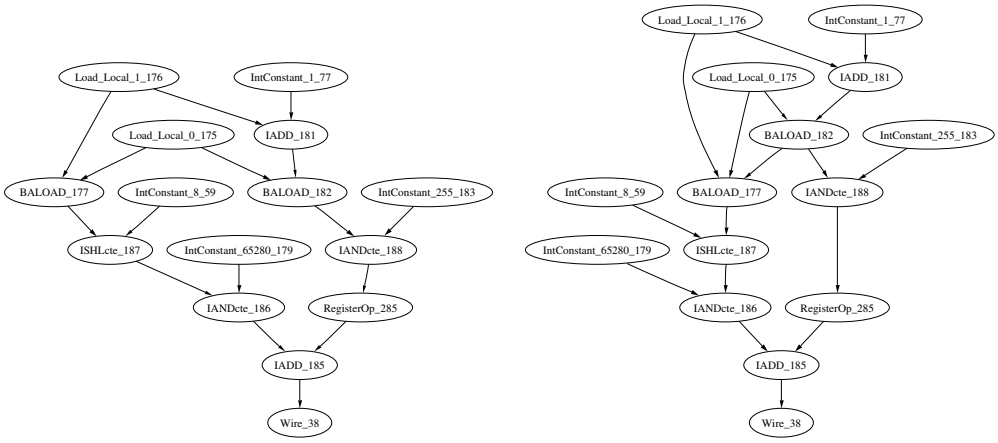
Listing 8.3: The *wordToBytes* method.

```
private static void wordToBytes(
  int      word,
  byte[]   out,
  int      outOff)
{
  out[outOff] = (byte)(word >>> 8);
  out[outOff + 1] = (byte)word;
}
```

Listing 8.4: The *mul* method.

```
private static int mul(
      int x,
  int y)
{
      if (x == 0)
        x = (BASE - y);
      else if (y == 0)
        x = (BASE - x);
      else {
        int p = x * y;
        y = p & MASK;
        x = p >>> 16;
        x = y - x + ((y < x) ? 1 : 0);
      }
      return x & MASK;
}
```

Finally we can see on Figure 8.2 the DFG and the sequencing graph of the byteToWord method. Those graphs have been generated by our compiler, using the dot format. The last '_' followed by a number, in each operation node, is simply a internal number given by the compiler.



(a) The DFG. It represents data dependencies between operations.

(b) The sequencing graph. It represents all dependencies between operations (time and data).

Figure 8.2: DFG and sequencing graph of *bytesToWord* method. Those two graphs have been generated by our compiler (dot format). The BALOAD operation is the operation that loads a byte from the memory. ISHL represents the shift left operation. The operation whose name is Wire represents the return operation (it is actually just a wire since it does not do anything). Other operations name should be explicit.

Notice that a register operation has been inserted to store the result of operation BALOAD_182, since operation IANDcte_188 does not store any result and since the result of operation BALOAD_182 is invalidated by operation BALOAD_177 (which shares the same resource). In fact all operations that ends with "cte" do not save their result, since they are not real operations but just wire connection (an AND operation whose one of its operands is a constant value can be simply implemented with just wiring). This is the result of an optimisation that we have implemented ; operation specialisation. This allow us to decrease the number of used resources and to decrease the length of the schedule (thus the execution time). Those operations are in fact pseudo-operations, since they do not respect the contract on the operations (Section 2.4.1).

8.3 Problems encountered

One of the main problem we have had with this IDEA algorithm is to make it fit into the FPGA[2]. The first attempt to generate the algorithm has failed because of the huge number of LE the algorithm was taken. Originally, we have not had support for method call in our compiler, so it is clear that the size of the algorithm, in term of resource inter-connection was bigger than this one, since we have had to in-line (by hand) all the method call.

Once we had implemented method call in our compiler, the problem was the routing of the resources. The routing is the task of connecting the different LE with buses. Unfortunately, there were still too many resources used and too many routes needed. Thus we have implemented clever algorithms to avoid inserting too many registers for instance, as described in Section 5.3.1. By example, at the very beginning of this project, we had not implemented the resources in such way that the result data was valid until its next execution, resulting in many registers inserted.

Finally we have spent many nights at launching the Quartus tools, in order to know if the emitted code corresponding to IDEA would be finally routed in the FPGA or not (it takes more than 20 minutes to have the answer !). We have tried many different options in Quartus, in order to find the ones that have finally successfully generated our coprocessor.

8.4 Resource usage analysis

In order to do the synthesis of the IDEA coprocessor, we have limited the number of multiplication operation to 1. All other resources have been set to unlimited number, except for the resource that communicate with the VMW interface, which is unique.

The total number of available LE in the FPGA is 4150. After synthesis, the number of used LE is 4253 (which is greater than the available LE, but as we will see, after fitting the count is different). 3720 are used for our coprocessor design (the rest being used by the VMW interface). 965 of those LE are used for the inter-connection logic between the resources and for the controllers. It represents 22% of the total area !

We can see on Figure 8.3 the number of resources and the total number of LE taken by those resources. Not all resources have been represented in this table, but only the ones that take the most LE. The World interface resource is the resource that does the communication with the VMW interface. It is quite big since it has support for operations such as writing byte into the

[2]Excalibur, EPXA1

Resource	Number	LE number
Registers	30	817
World interface	1	744
Multiplier	1	623
Adder	3	275
XOR	6	96

Figure 8.3: Number of resources used and number of total LE used after synthesis.

memory. Those operations, as we will see in Section 8.6.4 are very special and thus require more LE. The LE used by other resources such as the multiplier could be reduced by doing a pipelined version for instance.

We see that the bigger part consists of the registers that take many LE. On those 30 registers, 22 represents real variables. The other 8, are registers that have been inserted in order to preserve the validity of the result of an operation, as seen in section 5.3.1. Thus we see that if we want to improve the resource usage, we would have to find some variables that could be eliminated. As we will see in Section 8.6.3, it is possible to reduce the number of variables by doing some simplification on the control flow.

After the fitting step, we finally end with 4083 LE used (the optimisation parameters of the fitter has been set in order to balance between timing and area optimisation).

8.5 Performance comparison

Once synthesised, our IDEA coprocessor runs at a frequency of 18 MHz, which is no sot bad, since the hand-written coprocessor runs only at 6 MHz.

We can see on Figure 8.4 the comparison between the different versions of the IDEA algorithm (the results for the hand-written coprocessor and for the pure software version are taken from [3]).

Our original Java program is using byte array. Thus to read 32 bits of data, 4 memory read accesses are needed where only 1 is needed in the hand-written coprocessor. In order to compare apples with apples, we have written a version of our Java program that uses int array. Furthermore, as the synthesised coprocessor corresponding to the int version of the Java program uses only 32 bits access to the memory, the memory controller uses less LE, so we are able to achieve a better clock frequency (in order to compare the coprocessor corresponding to the int Java program with the one corresponding with the byte Java program, we have run it at a lower frequency than

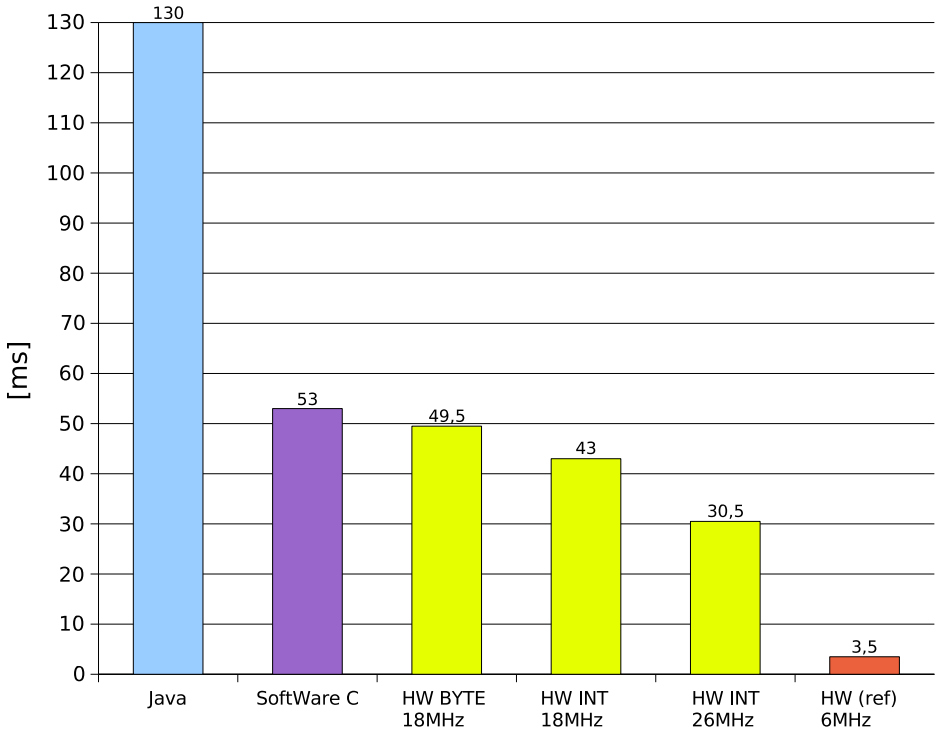


Figure 8.4: Performance comparison for the different versions of the IDEA algorithm. The input size is 8KB. The first measure is the pure Java version (using byte array), executed from the JVM. The next result is the pure software version (written in C, using short array). The 3 next measures are the hardware version of IDEA generated by our compiler (the first is using the Java program that uses byte array, while the two others use a modified version of the Java program that uses int array). Finally the last measure is the hand-written coprocessor.

it maximum allowed frequency). When running at the same frequency, we can see a gain of 6,5 ms by using the int Java program. This gain can be explained by the fact that we need to do more memory accesses when using byte than when using int. This is explained in section 8.6.4.

If we compare the Java program that uses byte and the corresponding coprocessor, we see that is is 2,6 times faster. But we have to take into account the overhead of the call (from the JVM side), which is about 18ms. Thus the real execution time is about 68ms, which is still about 2 times faster than the original Java program. The overhead of calling the coprocessor from the JVM is very high, it is due to the JVM and also to the native library. This overhead could be reduced by optimising the native library, and by modifying

the JVM, either to make it support built-in execution of the coprocessor or to reduce the overhead of calling native library methods.

Compared to the pure software version (written in C and using short array), our coprocessor is not so fast. If we compare now our coprocessor to the one written by hand, it is not as fast as it could have appeared compared to the pure Java program. The int version is about 10 times slower than the hand-written one.

Even if the results obtained seem to be bad, we have to make some remarks. First, the hand-written coprocessor is favorized ; the key is read at the beginning of the execution and is then stored in an internal RAM. One could argue that we are doing the same, since the first time the key is read, it is read from memory and the next value will then be retrieved from the cache of the VMW interface. But the fact is that even if the data is in the cache, it takes at least 4 clock cycles in order to retrieve it (due to the actual implementation of the VMW interface). Knowing that our coprocessor runs at 18MHz, and that we do 52 accesses to the key to compute 64 bits (the size of a block), the total number of key accesses is 53248 (we have 1024 blocks !). If we loose 3 clock cycles per access, we end with a total of 159744 clock cycles wasted, which represents 8,875 ms. This is 18% of the total execution time of our coprocessor ! For the int version that runs at 26MHz, this represents 6,144 ms, which is 20% of the execution time !

So to conclude this performance analysis, we can see that the way the Java program has been written directly affects the performance of the generated coprocessor. By rewriting the Java coprocessor in a clever way (or by choosing another implementation), we see a reduced gap with the hand-written coprocessor. On the other hand, a better compiler with optimisations could be able to "group" memory accesses that are lesser than 32 bits. But other aspects are not negligible, such as the ones explained in the next section.

8.6 Room for improvement

The IDEA example shows us a lot of places where we could improve the synthesis. We will see some of the optimisations that could be done (this is not an exhaustive list, only a brief overview of what would be possible to do).

8.6.1 Exploit more parallelism

Our compiler tries to do the maximum number of operations in parallel whenever possible. But we do this only inside basic blocks. In the case of

the IDEA algorithm, it would be possible to exploit more parallelism by doing loop folding for instance. This would thus create a pipelined version of the IDEA coprocessor. This could bring it closer to the hand-written coprocessor performance (the hand-written coprocessor has 3 pipeline stages).

8.6.2 Call optimization

As we have seen in Section 6.5, each time we call a method, we stop execution of the caller method. This has been done in order to simplify the compiler, but it is possible to improve this situation. Knowing which resources a called method uses, we could improve the degree of parallelism by doing, in parallel with the called method, some operations of the caller method. In order to improve a step further this optimisation, we could directly incorporate this notion into the binding step, in order to try to bind operations of the caller method to different resources than the ones used by the called method, whenever possible.

If we take a look at Listing 8.1, we can see that it would be possible to execute in parallel with the *mul* method calls some other operations such as reading the workingKey array in advance. If we consider that we do 3 accesses to workingKey in parallel with the *mul* method calls, and that one access take 4 clock cycles (this is the minimum), we would end with 12 clock cycles saved inside the inner loop. That is 98304 clock cycles for 8KB data, which represents about 4 ms at 26MHz (13% of the coprocessor execution time) !

Another aspect of method call can be improved ; the way the parameters are passed to the called method. As the parameters are passed by registers, it takes one clock cycle in order to "copy" them in the registers of the called method. We could do the things a little clever by directly using the parameters from the called method without going through the registers. Doing so would not require more resources than using registers to pass the parameters. The benefit of modifying the way parameters are passed could save us 34816 clock cycles for the calls of the *mul* method in our example, which represents 1,33 ms at 26MHz (more than 4% of the total execution time).

8.6.3 Control flow simplification

If we take a look at Figure 8.5 we will see the control flow graph corresponding to the *mul* method. The basic block number 200 does the $(x == 0)$ test and basic block 202 does the $(y == 0)$ test. If the first test succeed, basic block 201 get executed and if the second succeed, basic block 204 get executed. Else basic block number 205 is executed. The first thing we can notice is the

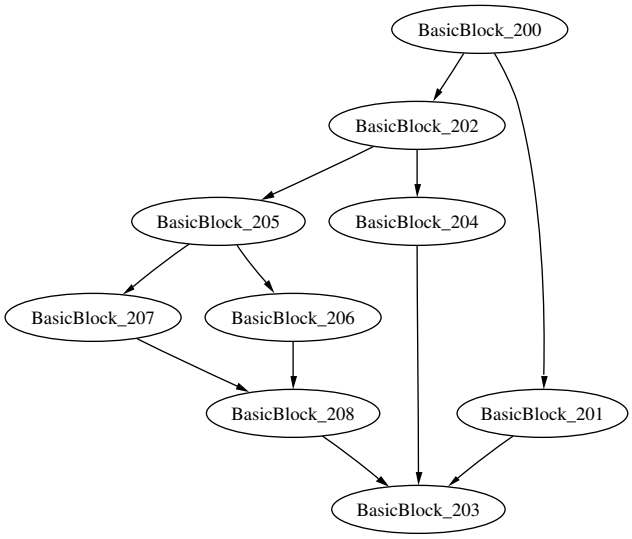


Figure 8.5: The control flow graph of the *mul* method.

fact that we could have done directly the two tests in a unique basic block, in parallel. This would have saved one clock cycle if the first test would have failed.

As we can see, the statements in the "else" branch of the *mul* method get subdivided into several basic blocks, due to the conditional expression $((y < x)?1 : 0)$. But in fact this is not necessary because the result of this expression could be expressed simply by some logic (the comparator operation returns 0 or 1 depending on the comparison). Doing such would have simplify the control flow graph and we would have save 2 clock cycles (one for the comparison and one for the STORE operation that occurs in each basic block, in order to store the value in a register).

Thus, by doing simple simplifications on the control flow, we could have saved 3 clock cycles in the *mul* method. As it is called 4 times in the inner loop of the idea algorithm plus 2 times outside, this would have saved 104448 clock cycles for 8KB input data ; more than 4 ms at 26Mhz (13% of the coprocessor execution time).

8.6.4 VMW interface improvement

One of the main performance penalty is due to the VMW interface and its impossibility to deal we something else than word access to memory (either read or write access). In the case of our IDEA algorithm, the original Java

program uses byte array. As the byte arrays are "packed" in the memory, when we want to do a write, we have to proceed in several steps :

1. we have to read the word at the address
2. then to do a mask with the value to write and the read value
3. finally we can write back the result

The first and the third step are costly, since we have to play a "ping-pong" game between the coprocessor core and the VMW interface.

This problem appears with the other type such as char and short. If the VMW interface had support for other access than word, it would improve the performance.

Chapter 9

Conclusion

We have seen along this project that synthesis of JBC is a complex task, involving many different steps in order to produce hardware. But, as compared with other assembly codes, JBC has some great advantages. Furthermore the concept of portability of Java programs is respected. Despite the fact that performance are poor for the IDEA coprocessor (relative to the hand-written version), we have actually shown that it was possible to do JBC synthesis. Some refinements needs to be done in order to achieve better performance, but the bases are now here to keep on with this work and to implement optimisations.

As we have seen, the theoretical gain that could be achieved by implementing some of the described optimisations could lead to a coprocessor whose execution time could be reduced by a factor of 2. The resulting coprocessor would thus be only 4 times slower than the hand-written IDEA coprocessor. This gap could be even reduced (and maybe filled) with the help of other optimisation techniques.

We have seen that the information contained in a Java method can be abstracted by the mean of several graphs. This kind of representation is not only adapted for the use by algorithms, but is also easy to understand and intuitive for a human. A good representation of the information is often a key component in a compiler design.

Resource binding is one of the big part of this work, since it affects the most the output of the compiler ; a binding that is not good means longer schedules and an increase in term of resource usage. In conjunction with resource binding, we have seen that the scheduling is another important part. Improvement in the scheduling could reduce the number of inserted registers, for instance.

The IDEA coprocessor that we have tried to synthesised during this project has given us a lot of trouble. Those difficulties have been all ad-

dressed. This has lead us to implement features such as method call, which were not in the original goals of this project. As we have seen, it is possible to encapsulate more than one method in the coprocessor, even if actually only one method can be called from outside of the coprocessor. The idea here would be to do the synthesis of a complete Java class. We could not only do the synthesis of methods, but we could imagine having the fields of the class directly in the coprocessor, in order to achieve better performance (like storing fields that represent arrays directly into the FPGA).

We could even see more ahead in the future and having only partial reconfiguration of the FPGA to better collapse with OOP languages (mechanism such as virtual dispatching could be integrated inside the coprocessor).

Finally, even if this project has taken me many nights, it was the most exciting one I have ever done. I would have liked to have more time to spend on certain parts of the work, but unfortunately 4 months were not sufficient. Nevertheless this project was very challenging and I do not have any regrets at all. I have learnt a lot about high-level hardware synthesis and compilation, and it was also the opportunity to learn more about reconfigurable computing.

Acknowledgements

I want to thank Michael Ruflin for his help during all my project (specially for the Java test program).

Cédric Gaudin and René Beuchat, who have kindly answered many VHDL questions.

Miljan Vuletic, for its support and advices.

And Finally all the LAP team, without whom this project would not have come to reality.

Bibliography

[1] C. Dubach, "Java virtual machine on fpga based platforms." Feb. 2004.

[2] A. Menghrajani, "Transparent pld use from java." Feb. 2004.

[3] M. Vuletic, L. Pozzi, and P. Ienne, "Virtual memory window for application-specific reconfigurable coprocessors," *Proceedings of the 41st Design Automation Conference, San Diego, Calif.*, June 2004.

[4] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[5] K. D. Cooper and L. Torczon, *Engineering a compiler*. Morgan Kaufmann, 2003.

[6] M. C. Golumbic, *Algorithmic graph theory and perfect graphs, 2nd edition*. Elsevier, 2004.

[7] G. De Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill, 1994.

[8] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek, "Object oriented circuit-generators in java," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.