Abstract—ASIP processors and programmable accelerators are replacing monolithic ASICs in more and more areas. However, the design and implementation of a new ASIP processor or programmable accelerator requires a substantial design effort. There are a number of existing tools that promise to ease this design effort, but using these tools usually means that the designer get locked into the tools a priori assumptions and it is therefore hard to develop truly novel ASIPs or accelerators. $\text{NoGap}$ is a tool that delivers design support while not locking the designer into any predefined template architecture. An important aspect of $\text{NoGap}$’s design process is the ability to design the data path of each instruction individually. Therefore the size of input/output ports can sometimes not be known while designing the individual functional units. For this reason we have introduced the concept of dynamic port sizes, which is an extension of the parameter/generic concept in Verilog/VHDL. A problem arises if the data path graph contains loops, either due to intra or inter instruction dependencies. This paper will present the algorithm used to solve this looping problem.

I. INTRODUCTION

The design and implementation of a new processor is usually the result of a substantial design effort. There are a number of different software tools (some of them mentioned in Section II) that relaxes the design effort in one way or another. However all these tools forces the designer into a predefined architecture template. This limitation in design flexibility often makes designers of novel ASIPs (Application Specific Instruction-set Processors) and programmable accelerators revert back to an HDL language, e.g. Verilog or VHDL. HDL languages offers full design flexibility at the register transfer level, but the flexibility comes at the cost of increased design complexity. All details, e.g. register forwarding and/or pipeline control, has to be handled manually.

Our aim is to address this problem, and for this reason we have instigated the $\text{NoGap}$ (Novel Generator of Accelerators and Processors) project\(^1\). $\text{NoGap}$ aims to give designers of novel ASIPs and accelerators a tool which will support them in their work. This support, however, does not come at the cost of sacrificing design flexibility or locking the designers into a predefined template architecture.

$\text{NoGap}$ approaches the problem of supporting while not limiting the designer in a novel way, $\text{NoGap}$ does not generate new accelerators or processors based on any template architecture and assumes very little about the system being designed. The underlying design principle in $\text{NoGap}$ is based on the assumption that designing individual modules, e.g. adders, multipliers or ALUs, of a pipelined architecture is generally a fairly simple task for a human, even for a relatively inexperienced designer. Specifying the temporal and spatial relations between these modules on a per instruction basis is also something humans are good at. The hard part for a human is to merge all these instructions into a pipelined ASIP architecture having the necessary multiplexers, control signals, and associated delays. On the other hand this is a fairly easy task for a computer since in principle no more creative decisions has to be made. For this reason the design input to $\text{NoGap}$ is descriptions of the individual Functional Units (FUs) (Functional Units) (can be supplied as a library), an instruction set\(^2\), the per instruction data path architecture, and a set of constraints. $\text{NoGap}$ then compiles this information to an intermediate representation that can be used to generate useful end products, e.g. assemblers, simulators, and synthesizable RTL code. This flow is shown in Fig. 1.

The requirement, that a designer shall be able to design indiviual components and describe their relations one operation at a time, introduces an interesting problem: port sizes might vary depending on where in the pipeline an FU is used. For example, in an eight bit system, one operation using an adder early on in the pipeline, will require eight bit inputs to the adder. Another operation uses this adder after a multiplication and then the input size has to be 16 bits. In short the exact port size depends on the overall pipeline structure and thus can not be known when designing a general FU.

This paper will present the method used in $\text{NoGap}$ to solve this problem. FU port size specification is done in a novel way and the algorithm used to solve this problem in a data path graph is presented.

The basic principles in $\text{NoGap}$ of further described in [1], [2], [3]

\(^1\)With the generous support from VR, the Swedish research council.

\(^2\)Fixed function data paths has an instruction set consisting of a single instruction.
II. RELATED WORK

A number of tools such as LISA [4], EXPRESSION [5], nML [6], MIMOLA [7], ArchC [8], and ASIP Meister [9], are tools that support processor design. All of these tools however force a designer into a predefined template architecture. On the other end of the spectrum of design tools are HDL languages such as Verilog, VHDL or SystemC [10]. These tools however require manual handling of all miniscule details of an RTL design. NoGap offers an unique trade off between these two extremes. No template design is assumed but support is given for managing details regarding pipelined instruction controlled architectures.

High level synthesizes tools such as Catapult-C [11] offers C to HDL generation for specific algorithms. These kinds of tools will make sure all FUs in the system have the correct port sizes. Although they are powerful tools for fixed function DSP hardware, they leave little room for designing efficient instruction controlled data paths.

Verilog and VHDL modules can be configured with parameters and generics respectively. These parameters must however be specified for each module instantiation and if its not entirely impossible to set these sizes as functions of what has previously been in the pipeline it is at least a very cumbersome and error prone task.

NoGap on the other hand, can dynamically determine the size of buses and ports. This extends the functionality already existing in Verilog and VHDL with parameters and generics respectively. While the parameterized modules can have their port sizes set from the outside, they can not determine their own port sizes depending on where in the data path they end up.

III. NoGap OVERVIEW

NoGap is a common name for a number of different components. The system can be divided into three main parts, NoGap\(_{CD}\), facets, and spawners.

NoGap Common Description (NoGap\(_{CD}\)) is an intermediate representation of the system being designed. NoGap\(_{CD}\) is generated from an Abstract Syntax Tree (AST) descriptions of the individual FUs. This AST description is constructed through a C++ API.

A facet is a tool that constructs a NoGap\(_{CD}\) through the C++ API. One facet has been implemented as a language, exposing all functionality of NoGap, called NoGap Common Language (NoGap\(_{CL}\)), this language is further described in [3]. A facet can also indirectly use the C++ API by generating a NoGap\(_{CL}\) description and rely on the NoGap\(_{CL}\) parser to generate the needed ASTs.

A spawner is a tool that reads the NoGap\(_{CL}\) to construct some useful output. For example, syntheziable Verilog code, a cycle and bit accurate simulator, or an assembler for a generated processor.

This layered approach allows for a flexible system, where spawners are independent of the facets. Using NoGap as a back end will make it easy to design a more dedicated facet, e.g. a DSP processor design facet or just a simple data path designer facet. The spawners will still work and generate the correct output.

For example, a cycle accurate simulator spawner could have been implemented for an earlier project but a new facet would ease the design effort for a new project. In this case only a new facet has to be implemented and time/money can be saved by reusing the old spawner.

NoGap\(_{CD}\) contains information about all FUs in the system. The two most important components of NoGap\(_{CD}\) are Mage and Mage.

Micro Architecture Structure Expression (Mage) is an annotated data flow graph describing all spatial and temporal connections between FUs for all operations defined on this particular data path.

Micro Architecture Generation Essentials (Mage) is a somewhat modified and optimized version of the original AST.Mage FUs are seen as black boxes in the Mage graph. Mage FUs can be seen as operators in Mage. For example simple FUs such as adders or multipliers, but also more complex Mage FUs can be used, such as ALUs or a complete MAC unit. It is up to the designer to decide how complex the Mage units shall be.

Note that NoGap does not treat storage elements in any special way. A register will be treated as any other FU. To inform NoGap about the possible source and destination operands, special instruction directives can be given in the FU used for decoder generation.

Listing 1 shows an example of NoGap\(_{CL}\) code describing one Mage FU (fu leaf) and Mage FU (fu operations), the Mage FU describes a number of operation a data path can perform. The exact syntax is not important for this paper. The sizing algorithm only depends on the Mage graph and the input/output port width expressions in the Mage and Mage descriptions. The reader is advised to note the format of the input and output size statements named auto.

In NoGap\(_{CL}\) port sizes can be set from a size expression as shown in Listing 1.

IV. DYNAMIC PORT SIZING IN NoGap

The dynamic port sizing in NoGap ensures that adding an instruction will not break already existing functionality even...
Listing 1: Annotated FU descriptions

```c
leaf //Fu declaration and name

ADD //All classes can be named.
input a_i[*], b_i[*]; //input and output ports...
input [2:0] b_i; //... with Verilog range.
output c_o[*]; //... or with dynamically configured ranges...

max(size(a_i[*]), size(b_i)) <= c_o[*];

@add: //Start of combinatorial block.
addr[0] = 0; //Note this clause name.
  c_o[*] = a_i + b_i;
  @else: SUB { //Also note this clause name.
    c_o[*] = a_i - b_i;
  }

@operations ADDS { //Operations can use the incoming bus size in both their input and output ports. The size of an output port can be written as a symbolic expression if no numeric value is available. The symbol (#), which we call incoming bus size, is used to denote the size of the bus coming into this port. Mage FUs can only use the incoming bus size for their input ports. This means that an output port size can be written as a function of one or several input port sizes. Two new operators can be used when specifying an automatic port size; binary maximum($) and binary minimum($@).

  if (op_a_i == 0) %ADD {
    l1.a_i = op_a_i;
    l1.b_i = op_b_i;
    l1.c_o[*] = 0;
    l1.RES = 0;
    @p1;
    res_o = l1.c_o[*];
    @p3;
  } else
    if (op_b_i == 0) %SUB {
      l2.a_i = l1.c_o[*];
      l2.b_i = op_b_i;
      l2.c_o[*] = 0;
      l2.RES = 0;
      @p2;
      res_o = l2.c_o[*];
      @p3;
    }

  //Final result is connected to the output port.

@pipeline the_pipe { //Declaring a pipeline...
  p1 -> ff -> p2 -> ff -> p3; //And its timing
}

@operation the_pipe { //Declaring an operation in the pipe pipeline.
  //Using %ADD as the instruction selection signal.
  if (1) and 12 will use its default clause (ADD) for their operation.
  operation { //The pipe} adds(instruction_i)
    operation { //The pipe} adds(instruction_i) {
      0p1; //Following assignments takes place in phase 1
      11.a_i = op_a_i; //Leaf fu 1 input c_i is assigned
      11.b_i = op_b_i; //... as is c_i
      0p2; //Following assignments take place in phase 2
      12.a_i = 11.c_o[*]; //12 input c_i gets the output from 11 output c_o
      12.b_i = op_b_i;
      0p3; //Following assignments take place in phase 2
      res_o = 12.c_o[*]; //The final result is connected to the output port.
      //Operation sub is almost the same as above
      operation { //The pipe} sub/Instruction_i {
        0p1;
        11.a_i = op_a_i;
        11.b_i = op_b_i;
        12SUB2; //Here is the difference, the SUB clause is used instead
        0p2;
        12.a_i = 11.c_o[*];
        12.b_i = op_b_i;
        12SUB1;
        0p3;
        res_o = 12.c_o[*];
      }
    }
  }
}
```

Fig. 2: Intra operation loop
Fig. 3: Inter operation loop

if the same FU is used by other instructions with hardware multiplexing. The sizing problem is complicated by the fact that a Mage graph might contain port size dependency loops, i.e. port size expressions that in one way or another depends on itself. The algorithm presented in this paper is able to handle these loops in a consistent manner.

The sizing algorithm is split into two main phases. First an annotation phase and then a size solving phase. The annotation phase annotates all sizes with either values or symbolic expressions if no numeric value is available. The size solving phase then uses the result from the annotation phase and resolves all dependencies, loops are resolved using a simple heuristic, described later.

Essentially loops in the Mage graph are either intra-operation, meaning that the loops are part of the data path for this operation, e.g. a MAC type operation would cause these loops. Or inter-operation, meaning that the loops are not part of any one operation, rather the loops are due to architecture artifacts occuring when some operations are merged into a single data path. Figure 3 shows an example of an inter-operation loop, where two operations, A and S are merged into one data path, A uses the adder first, then the subtractor, S uses the subtractor first and then the adder. Figure 2 shows an example of an intra-operation loop.

The sizing algorithm operates on Mage graphs, Mage FUs therefore needs to be connected in a Mage graph to be affected by the sizing algorithm. Mage graphs in turn might also have dynamic input port sizes, in this case the input port sizes are set to a symbolic values and all sizing computations must be done with symbolic expressions. GiNaC [12] is used to handle these symbolic computations. It was however necessary to extend GiNaC with two more functions; binary maximum($) and binary minimum($@).

V. THE SIZING ALGORITHM

The overall algorithm for sizing ports can be summarized as follows.

1) Make an initial annotation in the Mage graph of all edges to be sized, as described in Section V-A
2) Build a graph of all non trivial relations.
3) Reduce the graph for each non trivial symbol, as described in Section V-B
4) Set edge sizes in the Mage graph according to the reduced graphs.

The symbol (#), which we call incoming bus size, is used to denote the size of the bus coming into this port. Mage descriptions can use the incoming bus size in both their input and output port expressions. Mage FUs using dynamic input port sizing usually requires dynamic output port sizing as well, however Mage FUs can only use the incoming bus size for their input ports. The size of an output port can be written as a function of one or several input port sizes. Two new operators can be used when specifying an automatic port size; binary maximum($) and binary minimum($@), e.g. an adder would probably have the output port expression set to $(opa$opb) + 1, where opa and opb are the input port sizes. This approach presents two problems. First if one or several Mage inputs are dynamically sized, i.e. we dont know its size at compile time,
expressions, if the output node is automatically sized it should just propagate in the graph. Output nodes can result in new expressions. When processing an input node, the expression is multiplexing multiplexers are handled in the same way as FUs. Expressions will be set to a new symbol. Synthesized hardware FUs have their target expressions set. If not the empty target expression is set when processing input nodes. The algorithm processes the graph with a wavefront traversal. If the traversal reaches an FU input port, it checks if all other input ports of this FU have their target expressions set. If not the empty target expressions will be set when processing output nodes and target expressions are assigned two size expressions one called the source expression and the other the target expression.

Some extra care must be taken to handle inline expressions and inserted multiplexers. These concerns are outlined in Section V-1 and Section V-2, respectively.

1) Inlined Expressions FUs: In the \( \text{MaSap}^{CL} \) language, ports and signals can be assigned from expressions written directly in the operation construct, e.g:

\[
\text{fu}_a \cdot \text{dat}_i = \text{fu}_b \cdot \text{res}_o + \text{fu}_c \cdot \text{res}_o + 8
\]

The output size expression for this FU is automatically synthesized by \( \text{MaSap}^{CL} \). The sizing assumptions made for inline expressions are listed in Table I, where \( S(\alpha) \) means the size in bits of data \( \alpha \), \( \overline{C} \) is bitwise inversion, \( \ll \) is left shift, \( \psi \) is concatenation, \( \gg \) is logical right shift, \( \ggg \) is arithmetic right shift, \& is bitwise and, | is bitwise or, \( \oplus \) is bitwise xor, \&\& is logical and, \( \| \) is logical or, \( [\cdot] \) is bit index, \( \neg \) is logical not, and the other operators have their normal meaning.

2) Hardware Multiplexing Muxes: The synthesized multiplexers, used for hardware multiplexing of operations are assigned the output function of \( \max(S(i_0), S(i_1), ..., S(i_n)) \), where \( i_k \) is input port number \( \xi \).

3) Automatic Sized Signals: Signals used as part of a \( \text{MaSap} \) description FU can also be automatically sized and can be declared as

\[
\text{signal auto("#") sig_name;}
\]

A. Annotation Phase

In the annotation phase all edges in the \( \text{MaSap} \) graph are assigned two size expressions one called the source expression and one called the target expression. Source expressions are set when processing output nodes and target expressions are set when processing input nodes. The algorithm processes the \( \text{MaSap} \) graph with a wavefront traversal. If the traversal reaches an FU input port, it checks if all other input ports of this FU have their target expressions set. If not the empty target expressions will be set to a new symbol. Synthesized hardware multiplexing multiplexers are handled in the same way as FUs.

Processing input nodes does not result in any new expression, when processing an input node, the expression is just propagated in the graph. Output nodes can result in new expressions, if the output node is automatically sized it should have an output expression that depends on the sizes of some input nodes. The new expression for the output node is found by substituting the relevant input expressions into the output expression.

Figure 4 shows how this algorithm traverse a simple \( \text{MaSap} \) graph. The dotted line bubbles shows the different active node sets. The numbers in the active node sets show the order in which they are considered. Each edge has two dots a source and a target dot. If the dot is black is has been assigned a valid expression, if the dot is white is has been assigned a symbolic variable.

The annotation algorithm is presented in Algorithm 1. Note that this algorithm is stable in the presence of loops.

A small example of the resulting size expressions is shown in Figure 5, where text in square brackets represents the size of the signal and text in curly braces represent output expressions. The input ports of the adder are named \( i_1 \) and \( i_2 \), the input ports of the multiplexer are named \( A \) and \( S \). Remember that

\[
\alpha$/\beta$ = \max(\alpha, \beta).
\]

B. Solver Phase

When the annotation phase is completed all bus size relations are used to build a relation graph. The relation graph is built with two types of nodes, source and target nodes, for the
A target node can contain introduced symbols, an introduced symbol will, according to Algorithm 1, be equal to either a numeric size or a symbolic expression. If the symbol is equal to a numeric expression the symbol is directly substituted with its numeric value. If the symbol is equal to another symbolic expression, an edge is added from the symbol to a source node containing the symbolic expression. For each symbol in the source node an edge is added from the expression node to the target node for the symbol in question. An example of such a graph is depicted in Figure 6. Mathematically the graph can be described as Equation (1)–(2). Clearly, there is a loop in this example.

\[
\begin{align*}
S_7 &= f(S_6, b_i, c_i) \\
S_6 &= h(S_7, a_i, b_i) \\
S_7 &= f(h(S_7, a_i, b_i), b_i, c_i) \\
S_6 &= h(f(S_6, b_i, c_i), a_i, b_i) \\
S_7 &= f(h(0, a_i, b_i), b_i, c_i) \\
S_6 &= h(f(0, b_i, c_i), a_i, b_i)
\end{align*}
\]

The strategy for solving loops is to compute the maximum substitution for each function. The algorithm for finding the maximum substitution for a relation graph is given in Algorithm 2. All looped expression has to be solved simultaneously since hardware loops can be parallel and simultaneously dependent on each other. A simple example of this is depicted in Figure 9. After the algorithm has been run on the example graph in Figure 6 for symbol6 the new relation graph reached in Figure 10. When the algorithm is run for both symbol7 and symbol6, Equation (3)–(4) can be extracted from the graph. This algorithm will correctly handle complex cases. Such as a loop within a loop (Figure 8) and side loops (Figure 7). At this point there is no real solution since there are still self references in the set of equations. This is solved with a simple heuristic; assume that from the start all symbols starts a loop with no data and thus have a size of zero. Thus after the first iteration in the loop the sizes of \( S_7 \) and \( S_6 \) are according to Equation (5)–(6). Since all size expressions in the rest of the Mase graph is constructed from either constant input sizes, symbolic input sizes, or introduced symbols, substituting \( S_6 \) and \( S_7 \) with their looped expression will yield the correct result.

The algorithm presented here does only a single iteration in all looped expressions. It is desirable to be able to specify a maximum number of iterations that a connection should be used. The algorithm then needs to be modified so as instead of disallowing all substitutions of already substituted variables (this happens when a loop in the graph has been completed). A variable should be allowed to be substituted as many times as the minimum allowed iterations in a loop.

Please note that numeric sizes always take precedence over symbolic expressions. There will thus be no iteration calculations for numeric sizes.

**VI. Results**

NoGap has been used to reimplement a floating point adder/subtractor. The NoGap design was implemented in a couple of working days, with the added benefit of giving a clearer picture of the pipeline. Table II compares the NoGap implementation to Xilinx’ implementation.

<table>
<thead>
<tr>
<th>Device: XC4VSX-10</th>
<th>Xilinx</th>
<th>DA</th>
<th>NoGap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline depth</td>
<td>13</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>LUTs</td>
<td>578</td>
<td>846</td>
<td>876</td>
</tr>
<tr>
<td>FFs</td>
<td>594</td>
<td>429</td>
<td>533</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>368</td>
<td>290</td>
<td>217</td>
</tr>
</tbody>
</table>

**TABLE II:** Comparison with Xilinx adder in Virtex-4 [13].
A dual pipeline 16 bit RISC processor with DSP extensions called PIONEER has, been implemented in NoGap, the generated Verilog code was successfully synthesized and tested on an FPGA. The generated code met timing closure at 203 Mhz in a Virtex-4 speed-grade 12. Looking at the critical path, which was trough the adder/subtractor in the MAC unit, we can conclude that NoGap synthesized control paths are not the limiting factor.

The algorithm presented in this paper was used as part of the generation process in both of these implementations.

### VII. Future Research

At the moment the automatic sizing expressions are represented as strings and are as such separated from other part of the design, such as parameters. Future research will try to make the sizing expression a more integral part of FU design and a more natural part of NoGap. Listing 2 gives an example of NoGap code using new syntax to incorporate parameters into the sizing expressions.

It would be desirable to be able to use the input bus size symbol for sizing expressions in Mage, there are already functionality in NoGap for generating sizing expressions from individual data expressions, as described in Section V-1. There are also functionality for generating variable and expression dependency graphs. Putting these two techniques together would allow for use of the input bus size symbol in Mage fu output port expressions.

### VIII. Conclusion

This paper presented a method for data path design where the functional units in the data path can have port sizes automatically calculated depending on its placement in the data path. This paper suggested a syntax for a module interface descriptions in support of the dynamic port sizing. This scheme allows for symbolic data path input port sizes and thus symbolic expressions has to be used to resolve the port sizes of all FUs in the pipeline. In the end a module with symbolic input port sizes will have symbolic expressions for all sizes inside the module as well. Dynamic port sizes has been used in modules when designing various systems in NoGap and have proven to be a convenient feature helping but not limiting the designer.
IX. ACKNOWLEDGEMENT

The authors would like to thank Ching-Han Wang for his work on PIONEER. We would also like to thank Faisal Akhlaq and Sumathi Loganathan for their work on the assembler generator used to program PIONEER.

REFERENCES


