2010 Fifth IEEE International Symposium on Electronic Design, Test & Applications

Notations for Multiphase Pipelines

Christopher T. Johnston, Donald G. Bailey and Paul Lyons
School of Engineering and Advanced Technology,
Massey University,
Palmerston North, New Zealand
chris@johnston.geek.nz, d.g.bailey@massey.ac.nz, p.lyons@massey.ac.nz

***Abstract*— FPGAs, (Field-Programmable Gate Arrays) are often used for embedded image processing applications. Parallelism, and in particular pipelining, is the most suitable architecture for supporting the required high throughput. Although pipelining is a well known technique for hardware design and is simple to describe, our experience has been that people have many problems implementing working pipelines, especially for multiphase designs. Existing hardware description languages force developers to design pipelines as a special case of parallel architecture, which makes it difficult to ensure that the pipeline has internally consistent timing. This is especially problematic in multiphase pipelines. This paper shows how many of these problems may be overcome by basing the notation on sequential dataflow, and discusses control issues of priming, stalling and flushing, with a proposed compiler implementation.**

Keywords-component; FPGA, Visual Languages, Hardware Description Languages

I. INTRODUCTION

FPGAs often run at a lower clock rate than conventional microprocessors; instead they rely on multiple parallel functional units to achieve a higher throughput than microprocessors with higher clock speeds. In FPGA hardware design, there are two main parallelisation techniques to increase the speed of an algorithm. One is to send different data sets to a number of functionally equivalent hardware units operating in parallel. Each unit interfaces to local memory to minimise access conflicts. The other is to map operations onto pipelines.

Pipelining, combined with image streaming, is well suited to embedded image processing because it reduces the memory access requirements by converting an image's spatial parallelism to temporal parallelism. To be effective for two-dimensional image processing, stream processing requires custom data caching to maintain data between rows. Pipelining also reduces the propagation delay of each stage, increasing the maximum operation frequency of the application.

Although pipelining is a well known technique for hardware design, and is simple to describe, our experience has been that people have many problems designing pipeline algorithms and converting them into actual pipelines without errors. This is especially true for multiphase designs in which the clock rate is a multiple of the input data rate.

Different HDLs (hardware description languages), VHDL [1], JHDL [2], Verilog [3], Handel-C [4], express parallel operations in different ways, but all force developers to design pipelines as a special case of a parallel architecture, without guaranteeing that the pipeline has internally consistent timing. This requires the developer to manually check that pipeline operations are scheduled correctly. Some IDE tools such as Quartus II [5] support the checking process by showing designs as schematic diagrams. Hardware compilers, such as SA-C [6], remove many of the pipeline difficulties by directly mapping a software design to hardware. This approach removes major decisions from the developer, reducing the degree of control offered to the developer.

This work on pipeline representation is part of a larger project [7] called VERTIPH. Several different options for expressing pipelining were discussed in [8]; here, we expand on that discussion by justifying the use of a visual language over text-based languages for multiphase designs. We present a new visualisation for multiphase pipelines. We also explain how an appropriate representation of pipelines aids in the control of priming and flushing of pipelines.

II. PIPELINING REPRESENTATIONS

We contend that it is more appropriate to treat pipelines as a special case of sequential operations than as a special case of parallel operations.

Pipelines have certain similarities with, and share certain problems with production lines. An automotive production line could be used to assemble a single car at a time, but works far more efficiently if all its assembly stations are active simultaneously. This simultaneity of operations does not apply to individual vehicles. Instead, each car is assembled in a sequence of discrete operations. In designing such a system, it is the sequence of operations that matters, not their simultaneity. The same is true of data being processed in a hardware pipeline; as far as the data is concerned, the pipeline is sequential, transforming the data from input to output through a sequence of operations. Each processing element in the pipeline, like its production line counterpart, operates sequentially on each input data element before passing the results on to the output. It is only the

978-0-7695-3978-2/10 $26.00 © 2010 IEEE
DOI 10.1109/DELTA.2010.29

correct synchronisation between successive (serial) stages that allows each stage in a pipeline to run in parallel with all the others.

From the developer's perspective, it is less important to visualise the parallel nature of the operations than to visualise how the data progresses through the design. A pipeline will only operate correctly in parallel if the data supply is correctly synchronised with data consumption. When the parallelism of the operations is the chief feature of the notation, the synchronisation between stages is implicit (through shared variables or registers), requiring secondary notations (comments) to explain the timing. Therefore, we consider that it is more important to show the algorithm as a sequential process, while still highlighting that the operations run in parallel.

Algorithms' pipelines can become more complex when there is more than one datapath, such as when pipeline stages contain their own internal parallel operations. Since HDLs make no distinction between parallel and pipeline architectures, the resulting ambiguity can cause confusion. Consider the following Handel-C [4] pipeline; the statements are inside a *par{}* statement block to indicate that they run in parallel:

```
par{
  a = a +1;              //stage 1
  b = a << 1;            //stage 2
  c = b – 3;             //stage 3 two parallel
  d = b + 3;             //operations
  e = d << 1;            //stage 4 two parallel
  f = d + 2;             //operations
  g = e * f + c; //stage 5
}
```

These seven statements make up a complex five-stage pipeline. Stages 3 and 4 operate in parallel and the final stage receives data from the two stages before it. The structure of the Handel-C code can take some time to understand fully, but it is easy to see from a simple dataflow diagram, shown in Figure 1, that the output from block c needs an additional register in parallel with blocks e and f so that block g will receive correctly synchronised data. Such errors are not readily apparent in the parallel representation.

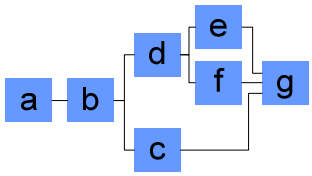

Figure 1. Graphical representation of the pipeline

In image processing, priming and flushing can be quite complex as significant work is often required to initialise look up tables, to prime filters to cope with edge effects and other data that the pipeline uses. The pipeline also needs to be flushed once new data has stopped arriving so that the data still in the pipeline is processed through to the output. Again, the designer needs to deal with this explicitly.

These aspects relate primarily to the sequential nature of pipeline behaviour. Treating pipelines as a type of parallel architecture only allows the designer to specify which modules are active simultaneously, but does not facilitate any of the more complex decisions referred to above. Treating a pipeline as a sequential system exposes the timing decisions that are such an important aspect of designing reliable pipelined image processing algorithms. Practical consequences of this argument are discussed in section 4.

III. MULTIPHASE REPRESENTATION

The sequence of events implied by the textual representation is lost with multiphase pipeline designs, where the clock rate is a multiple of the input data rate. This can be illustrated with an example five-stage pipeline (here in Handel-C) that is considered first as a single phase and then as a multiphase design. This example is part of an object tracking system described in [9]. Expressions have been represented by function calls to make the logic easier to follow:

```
if(pixels_data) par {
  rgb_to_yuv( rgb, yuv );      //stage 1
  threshold( yuv, label );     //stage 2
  filter_vert (label, vf );    //stage 3
  filter_horiz( vf, filtered ); //stage 4
  bounding_box( filtered );    //stage 5
}
```

Each of the five functions takes one clock-cycle and forms a stage in the pipeline; each function accepts data from the previous function and passes data to the next function block. The developer must infer this aspect of the design from the data dependencies. For a single-phase pipeline of moderate length, this type of representation, although hardly ideal, is workable.

To implement a multiphase pipeline, the different stages need to be separated either by conditional statements (Handel-C) or by incorporating separate modules with differently phased clocking (VHDL, Verilog). A two-phase Handel-C version of the object tracking algorithm is shown below. The order of the stages, and the implied dataflow, is no longer obvious because each phase is only clocked on every second clock cycle.

```
if(Pixels_to_process){
  if(phase1) par {
    rgb_to_yuv( rgb, yuv );      //stage 1
    filter_vert( label, vf );    //stage 3
    bounding_box( filtered );    //stage 5
  } else par {
    threshold( yuv, label );     //stage 2
    filter_horiz( vf, filtered ); //stage 4
  }
}
```

This makes it harder to follow the dataflow and makes modification and debugging difficult. Developers have to rely on comments or stepping through the algorithm each time a modification is needed, to avoid introducing data dependency errors. This is especially the case when adding a stage early in the pipeline; the phasing of all subsequent stages must be adjusted.

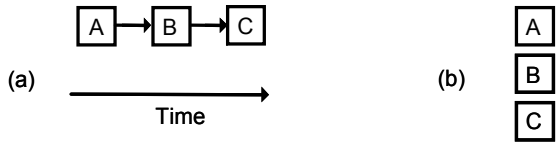

Figure 2. Sequential operations (a), Parallel operations (b).

The representation used for pipelines should make it easy to manage this data dependency, which is a consequence of the sequential characteristics of the pipeline and not the pipeline's ability to process data in parallel. With multiphase pipelines an ideal representation would also show when new data arrives into the pipeline, so that the pipeline phasing can be determined. Programming such systems with a textual notation is difficult; they can be extended to provide a representation of parallel systems, but they have difficulty representing systems that have a mixture of sequential and parallel components. The two dimensions of visual notations can readily separate sequential and parallel operations, such as in Figure 2 in which (a) shows a sequence of operations as a horizontal row and (b) shows parallel operations stacked vertically

Our visual notation for pipelining reinforces the sequential nature of a pipeline. This updates the pipelining notation discussed in [8] based on comments from a formal user evaluation [7]. The previous example is illustrated first as a single-phase pipeline in Figure 3 and then as a two phase pipeline in Figure 4. The components of the diagrammatic pipeline representation and its layout have been chosen to provide an intuitive insight into the semantics of a pipeline. Figure 3 illustrates the six main components. First, an L-shaped *if* control encloses the pipeline, and controls when the pipeline Second, there is a set of triangular input ports. Third is a set of rectangular boxes containing the operations. Fourth, dashed vertical lines show the clock cycles. The fifth is the set of horizontal rectangles at the top representing how the data moves thought the pipeline. Sixth, is a slider for modifying the data clock rate.

In Figure 4, the three rectangles indicate that the pipeline needs three pieces of data to be primed, although the pipeline still has five processing clock cycles, the data is output after 2½ data clock cycles The only difference between the single-phase and two-phase deigns is the phasing control slider and the generated pipeline bars.

A visual notation is not necessarily needed. A textual HDL such as Handel-C could be extended to provide similar control (below). Where the 2 in *pipeline(){}* indicates the multi-phasing rate and the <#> the functions stage order. The text notation does not provide visual feedback but is compact.

Functions that take more than one clock cycle but are within the data-clock period can also be put in the pipeline. Such operations that span multiple processing clock cycles are easier to visualise in VERTIPH than text, though are currently not supported in the present prototype.

```
if(pixels_data) {
 pipeline(2){
  <1> rgb_to_yuv( rgb, yuv );
  <2> threshold( yuv, label );
  <3> filter_vert( label, vf);
  <4> filter_horz( vf, filtered );
  <5> bounding_box( filtered );
 }
}
```

IV. PIPELINE CONTROL

The representations shown in Figure 3 and Figure 4 show the normal operation of the pipeline once it is loaded with data. Further control logic is required to handle the boundary cases (when data starts or stops). In stream-based image processing, these typically occur at the start and end of each row of data, and at the start and end of each frame.

There are three states other than normal pipeline operation: a priming state, a flushing state and a stalling state. The particular control mechanisms and state selection depend on whether the timing is source-driven (for example, data streamed from a camera) or sink driven (for example, data being streamed to a display).

While the pipeline is being primed, it is necessary to ignore the invalid data the pipeline produces. For source-driven pipelines, this may simply be a matter of producing a data-valid token when valid data reaches the end of the pipeline. However, the process is a little more complicated when producing pipelined versions of filters. Such pipelines must cache information from a two dimensional field, as the filter must accumulate data from several rows and columns in the image before it can produce valid output (the filter vertically box in Figures 3 and 4 uses row buffers to cache input data from previous rows). This implies that each block may require additional control signals to indicate the start and end of line and start and end of frame to enable correct initialisation and determine when output data is valid. For cases where the structure is regular, such as filters, the logic may be created automatically by the compiler, but more complex cases will require the user to provide additional pipeline control logic.

For sink-driven pipelines, it is necessary to preload the pipeline with valid data so that data will be ready on the output when required. For simple cases, this involves starting the pipeline at some predetermined time before the output data is required. In more complex cases, it may be necessary to start the pipeline as soon as data is available at the input, and then stall the pipeline once it is primed and valid data is ready on the output. Stalling keeps the data in the pipeline while waiting for new data. This is accomplished by not clocking the pipeline until new data arrives or a trigger condition is met.

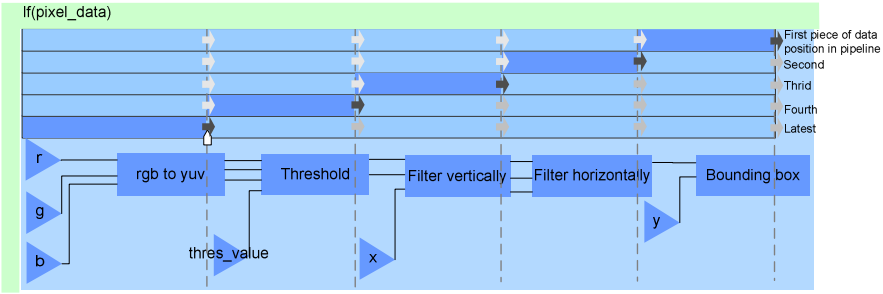

Figure 3. Single phase pipeline representation

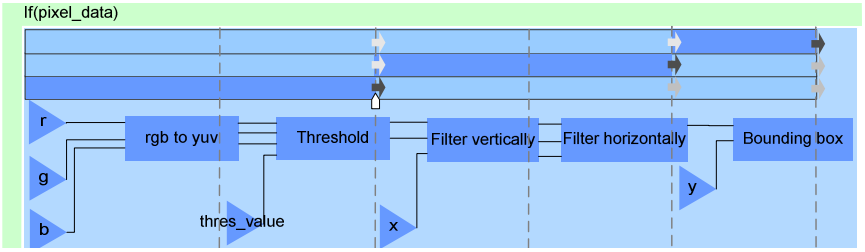

Figure 4:. Two-phase pipeline representation

At the end of each row or frame, additional control logic is required to flush the pipeline. For source driven processes, this requires running the pipeline for several clock cycles after the last valid data has entered, until all the valid data has passed through the pipeline. Again, with filters the control becomes more complex because they may require running the pipeline for several rows after the last image row is input. With appropriate caching, the stages before the filter may not need to run (further complicating the control).

For sink driven processes, it is only necessary to continue running the pipeline while output data is required (assuming that the width of the output image is less than or equal to the width of the input data). When the end of row is reached, priming may begin immediately for the next row.

Defining these control activities is a fruitful source of errors. By having an explicit representation for a pipeline, much of the logic to control the pipeline may be inferred automatically by the compiler. This is particularly the case with the normal operation of both standard and multi-phase pipelines. For one-dimensional pipelines, the control logic for priming and flushing both source and sink driven pipelines can also be inferred.

With two dimensional pipelining, the control logic is more complex because it must also take into account cache management. These may require a high-level event-triggered and state-based controller (such as that described in [11]) to provide the auxiliary control signals. While the developer must still design the complex priming and flushing logic, this will be made easier by having an explicit representation of the pipeline that models its sequential behaviour.

Using the graphical notation, the pipeline can be extended to have the three other states associated with it. Any special operations can be added in these states. In general they can default to setting a data invalid bit until the pipeline is primed, running the pipeline until flushed using zeros for data input and stopping operation of the pipeline during a stall operation. These states can be hidden and only made visible when required. A similar approach can be taken in text where each pipeline could have a *prime{}, flush{}* and *stall{}* control, similar to the required fields in a class definition.

By having a notation specifically for pipelining, and grouping the components that make up a pipeline together, the developer can be presented with a clearer representation of the design.

V. IMPLEMENTATION ON HARDWARE

While the prototype of VERTIPH has concentrated on the design from the developer's point of view, the FPGA implementation of the pipeline has been considered. The control for this is a one-hot token passing design similar to that used for serial control in Handel-C [12]. To run a single phase pipeline, both the clock and the state of the pipeline are important. Once primed all stages are clocked and run parallel. While priming, the operations in each stage can be enabled in succession, like sequential operations but with the exception that they do not turn off. When the pipeline is stalled, the clock is disabled for all operations. When flushing, each state is turned off as the last piece of data leaves. Figure 5 shows the priming, operational and flushing stages. Multiphase operation is not significantly more complex as the operations still become active as data fills the pipeline and then all run in parallel. However, they are running at a higher clock rate than the data rate which means that the operations need to stop when there is no data. This extra control can be thought of as a token for the new data that propagates with the new data as it moves thought the stages, as illustrated in Figure 6.

Branching can be accommodated through the use of conditional statements (*if-then-else*, *switch*). If the conditional operation is too long to be completed in a single clock cycle then it needs to be manually scheduled and pipelined by the developer to produce the appropriate condition at the correct time.

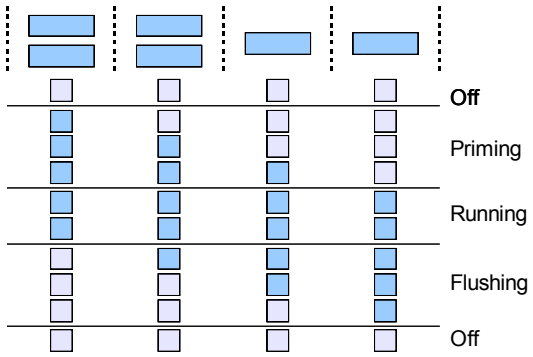

Figure 5. Token passing for control (blue on)

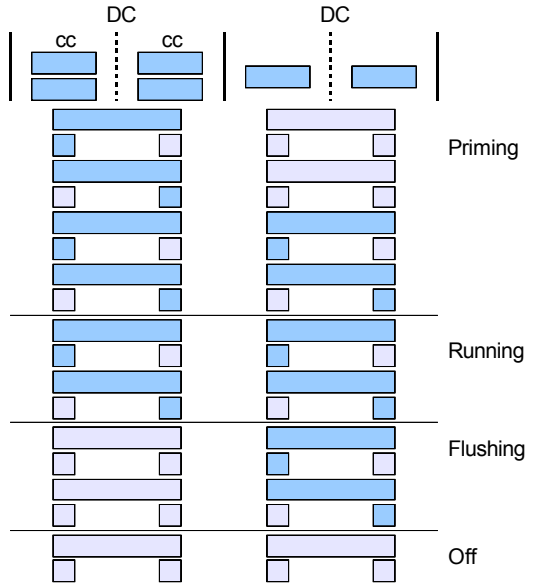

Figure 6. Token passing for control of multiphase pipeline

VI. Discussion

We have explained why pipelining and parallel operations should be treated separately to aid designers. By treating pipelining as a special case of sequential operations it is possible to produce a visualisation of the design which provides more information to the developer than the parallel representations which are currently prevalent in the hardware design space. This has resulted in a pipeline representation that is based on the sequential dataflow of expressions to which external pipelining information has been added.

Two extra features distinguish pipelines from sequential dataflows. First, a pipeline graphic has been added to indicate that several sets of data can enter the datapath on successive clock cycles and be processed simultaneously. Secondly, a phasing control slider-bar is incorporated, to allow the designer to specify how often data may legally enter the pipeline and to indicate the latency with respect to the input data rate. This updated pipeline notation makes the pipeline's datapath clear, and illustrates the data flow between expressions. It also makes the distinction between parallel and pipelined stages clearer. The extra pipeline-specific information is shown in a way that provides more information about the pipeline without detracting from the expressions which make up the pipeline.

A method to implement the pipeline is proposed using a modified one hot state machine. This is still in the design phase and is yet to be implemented in the current prototype for VERTIPH.

References

[1] Accellera, EDA Industry Working Groups - VHDL,http://www.vhdl.org/, visited March 2008.

[2] Brigham Young University, JHDL, www.jhdl.org, visited on 21 February 2005.

[3] IEEE, IEEE Standard Verilog HDL, http://www.verilog.com/IEEEVerilog.html, visited August 2008.

[4] I. Alston and B. Madahar, "From C to netlists: hardware engineering for software engineers?" Electronics & Communication Engineering Journal, vol. 14, no. 4, pp. 165-173, 2002.

[5] Altera, Quartus II http://www.altera.com /literature/lit-qts.jsp, visited February 2008.

[6] B. Draper, W. Najjar, W. Bohm, J. Hammes, B. Rinker, C. Ross, M. Chawathe, and J. Bins, "Compiling and optimizing image processing algorithms for FPGAs", Proceedings. Fifth IEEE International Workshop on Computer Architectures for Machine Perception, pp. 222-231, 2000.

[7] C. T. Johnston, "VERTIPH: A Visual Enviroment for Real Time Image Processing on Hardware", PhD Thesis in Computer Systems Engineering, School of Engineering and Advanced Technology, Massey University, Palmerston North, 2009.

[8] C. T. Johnston, D. G. Bailey, and P. Lyons, "Towards a visual notation for pipelining in a visual programming language for programming FPGAs", 7th International Conference of the NZ chapter of the ACM's Special Interest Group on Human-Computer Interaction, pp. 1-9, 2006.

[9] C. T. Johnston, D. G. Bailey, and K. T. Gribbon, "Optimisation of a colour segmentation and tracking algorithm for real-time FPGA implementation", Proceedings of Image and Vision Computing New Zealand, Dunedin, pp. 422-427, 28-29 November, 2005.

[10] J. R. Schermerhorn, Management, Sixth ed. New York: John Wiley & Sons, 2001.

[11] C. T. Johnston, P. Lyons, and D. G. Bailey, "A Visual Notation for Processor and Resource Scheduling", IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2008), Hong Kong, pp. 296-301, 23-25 January, 2008.

[12] I. Page and W. Luk, "Compiling Occam into field-programmable gate arrays", Proceedings of the Field Programmable Logic and Applications, Oxford, UK, pp. 271-283, 4-6 September, 1991.