ΨP periodica polytechnica

Electrical Engineering
52/3-4 (2008) 197–208
doi: 10.3311/pp.ee.2008-3-4.08
web: http://www.pp.bme.hu/ee
© Periodica Polytechnica 2008

RESEARCH ARTICLE

Pipeline mode in C-based direct hardware implementation

Péter Arató / *Bence* Csák

Received 2007-12-09

Abstract

In this paper a methodology is presented that enables the pipeline function of hardware blocks created by C-based direct hardware design. The method is embedded into the C-based design methodology worked out by the authors earlier. This pipeline enabling method is rather flexible, needs no special efforts. With the help of a simple state-machine-based entity, blocks of different execution times can build up the pipeline, even with data-dependent duration. A data-spreading technique solves data consistency. Pipeline sectioning – chosing the right and balanced granularity versus pipelining overhead – is an optimisation matter. Simulation results prove the correctness of the method.

Keywords

C-based design · reuse · component based design · parallel processing · pipelining · hardware-software codesign

Acknowledgement

The authors work has been supported by the grants OTKA Nr T043329 at the Department of Control Engineering and Information Technology, BME

Péter Arató
Bence Csák
Department of Control Engineering and Information Technology, BME, H-1117 Budapest Magyar Tudósok krt. 2., Hungary

Introduction

There is a big pressure on companies in recent times to use application oriented components. The reason is that there are more and more high speed applications, which require very high speed processing of complex algorithms. Such application is for example on-line video filtering, or wireless communication protocols. Of course the problem can be solved with a CPU-based, SW controlled circuit, but if e.g. heat dissipation is a leading constraint, distributed processing in FPGA or ASIC can be more beneficial. If the design is RTL based, then any modification needs high efforts and specially educated developer. Contrary, C-based design provides an easy way [1,2,3,4,5].

C-based design however has a strong sequential influence due to the initial sequential code. Better utilisation of processing time and silicon area requires – if possible – parallel processing.

Parallel processing – depending on the nature of the task – can be classified into two main types, one is „horizontal" and the other is „vertical". The first, trivial case is, when calculations are independent, so these can run partly or entirely in a concurrent manner. The other case is, when there is a sequential dependency among calculation parts, so these can not overlap. In case of cyclic execution, pipelined calculation scheduling can establish a so called „vertical" parallel processing.

In the first part of this paper the authors' embedding methodology is introduced, showing an example for „horizontal" paralleling in an application, where a recursive function is implemented. Concurrent operations compete for a single-sample resource and a resolving method is introduced.

In the second part the add-on methodology is introduced, which solves pipelining.

The embedding methodology

The pipelining solution discussed in this paper is embedded in the C-based direct hardware implementation methodology worked out by the authors earlier [8,9,10,11]. This embedding methodology is based on state-machines that correspond to basic C instructions like 'while', 'do', 'for', 'if', 'switch', or 'case'. These state-machines (instruction units) control corresponding actions like evaluation of conditions, starting cycles or other actions based on the result of the evaluated conditions and triggering subsequent operations. There are so many types of instruction units, as many instructions exist. Procedure ('function') definition - with its variables – is done by creating a procedure unit consisting of a state-machine and storages for local variables and calling parameters. Operations like comparison, addition, etc are represented by operation units. All these units form a compatible set of IPs, which then – as defined by the SW code - are linked into a chain using 'trigger' and 'ready' connections. We have proposed to break this strongly sequential structure with automatic compile-time code analysis based paralleling, allowing and resolving competition cases [11]. This automatic analysis is done by a wide-scope parser that – in addition to usual parser functions - analyses big blocks of code and able to recognise situations where paralleling or resource sharing is possible. It could be fed back from the resulting chip plan, so it could find a trade-off between resource demand and execution time. Thank to these basics, this conversion methodology is simple, yet structured and transparent, supporting creation of applications of any complexity. The distributed control helps better utilisation of silicon area.

The conversion method has a dual nature. Based on the C code, directly a HW structure can be drawn (so this is a direct conversion procedure). But - as text-to-text conversions require less apparatus – a C-to-VHDL methodology is given too. Here, block schemes will be shown for the easy understanding. All the blocks have a reset and a clock input, which are omitted in the figures. Outputs are changed at rising clock edges, while inputs are read and state variables are set at falling clock edges. This provides a half clock period for optional asynchronous circuits to settle.

On Fig. 1 the hardware block of instruction 'for' and its state diagram can be seen. A 'for' instruction in the form of "for (scv(); chk(); rcv())" is considered here, where function scv() is responsible for setting the cycle variable, chk() is to check, whether the cycle is to be executed and rcv() is to refresh the cycle variable. As in C description, one or more of these functions can be omitted. In such case the corresponding trigger has to be directly connected to the returning ready.

Signals on the right side of the block can be ordered into three groups. The first two signals belong to "scv()" and are responsible for setting the cycle variable. The next three signals belong to "chk()" and are responsible for checking the cycle condition, while the next two signals are responsible for triggering the cycle body and taking control back from the cycle. The cycle body involves steps corresponding to "rcv()".

The state diagram on the right describes the exact function. Rounded rectangles represent each state. Arrows between them represent possible state transitions. Unframed texts at the arrows show the condition of the corresponding state transitions (if there are multiple equations, a logic AND is considered between them unless it is explicitly noted there). Framed text at each status show, what actions are taken there.

Earlier publications [8,9] show examples of the embedding methodology; here another example is given where flexibility of handling function calls and returns - including recursion - is demonstrated. Parallel call of the same function sample is resolved by competition management.

The following SW is not a practical one, but helps highlight features mentioned in the introduction.

```
char func_r(char num)
  {
   if (num == 68) return 86;
   if (num >  12) return 13;
   if (num >   1) return func_r(num-2); //
 recursive call of func_r()
     else return num;
  }
void main(void)
  {
   char a,b,c,d;
   a = func_r(68);
   b = func_r(14);
   c = func_r( 7);
   d = func_r(68);
  }
```

Function "func_r()" has more features, which are important regarding the embedding methodology. It has more than one "return" statements, that are not mutually exclusive and one can be even recursive. The recursive operation determines whether the parameter (if num < 13) is an even or an odd number.

Multiple returns in a C function – even if these are non-exclusive – are usual and the flow-control resolves them, but since the base methodology supports concurrent execution, all the returns provide a value, which situation could not be resolved. Consider, if "func_r()" is called with 68, then all "if"s are true and the function would return with 86 and 13 and – in addition – it would call itself again with 66.

The solution to this problem is a "return" unit, of which so many is needed as many "returns" are in the source code. These are then linked into a priority chain. Each return unit latches the corresponding return value, but finally only the one with highest priority may output its latched value. Using this block, even functions not explicitly having any "return" statement can be converted, if during the C to HW translation one such block is inserted. Fig. 2 shows the block scheme and the state diagram of the "return" unit.

The "return" unit gets its return parameter on its "in" input. When "trg" is activated, the value is written into an internal register and "rdy" is activated. Return units are daisy-chained by priority ranking using their "dsyi" inputs and "dsyo" outputs. If a "return" unit has been written, it raises its "dsyo" output. The same happens anyway, if its "dsyi" input is raised by a higher priority "return" unit. This chain works asynchronously and a half clock cycle is provided for full propagation. When later the

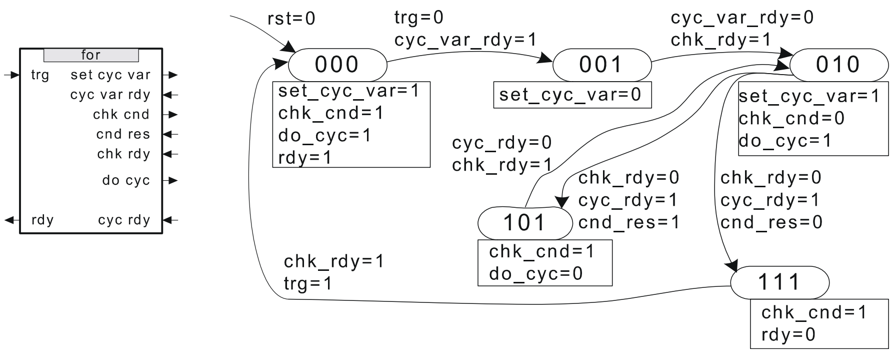


Fig. 1. Block scheme and state-diagram of the state-machine representing instruction 'for'

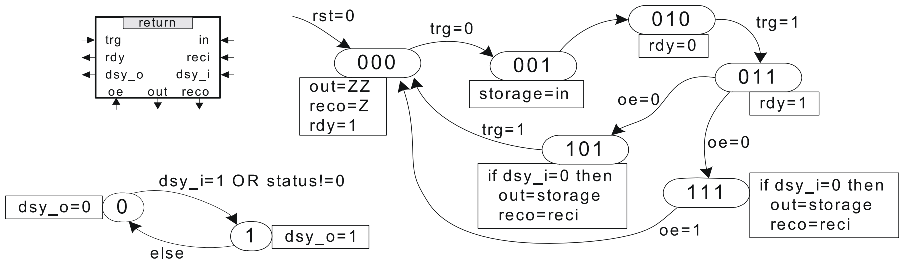


Fig. 2. Block scheme and state diagram of the "return" unit

"oe" input is activated, the single "return" unit with dsyi=0 and dsyo=1 drives its "out" and "reco" outputs. "Out" contains the stored input value, while "reco" is set according to the "reci" input. "Reci" stands for "recursion demand in" and tells whether the respective return path is a recursive path (like the "return" of the last "if" in "func_r()"). When "oe" is inactive, these outputs are in high-impedance state ("Z").

For supporting recursion, another unit is required, named "recurs". It either transfers calling parameters to the sample of the function and then transfers the return value back to the caller, or – if the function itself signals a recursion demand – it feeds back the return value as it was a new calling parameter and restarts the calculations of the function. Block scheme and state diagram of "recurs" can be seen on Fig. 3.

The "recurs" unit is a function header block, transferring control signals and data between operations of a function and the calling blocks. It gets the calling parameter(s) on its "parin" input(s) and is triggered by its "trg" input. Then it outputs the value of "parin" to its "paro" output and triggers the first function operation by its "trgo" output. When operations of the function are done, the last puts the return value to "retvali" and then triggers "rdyi". If "reci" is inactive, then it is a non-recursive call, so the value of "retvali" is output to "retvalo" and then "rdy" is activated to signal that the function is done. If "reci" is active, then recursion is needed, so the value at "retvali" is fed back on "paro" and the operations of the function are restarted via "trgo".

Note, that this recursion mapping inherently assures a faster realisation of the recursion, since there is no stack used and so no stack restoration is needed.

Using these and some other blocks, the following block scheme represents „func_r()" on Fig. 4.

As on Fig. 4 can be seen, the blocks are chained as determined by the original C source code using trigger and ready connections. For saving space, paralleling is not introduced here, but later in connection with the „main()" program.

The „main()" program shows further features of the basic methodology. These are: using functions in „manager-resource" approach, parallel execution and competition resolution.

The „manager-resource" approach enables the reuse of resources like function samples, array variables, operations, etc. The resulting hardware can consist of one single resource of a kind (e.g. func_r() or a multiplier) and as many corresponding managers as many times the resource is „called" in the original source code. A bus system is created for the access of each resource to which the corresponding managers are attached [10].

The basic methodology inherently supports paralleling since one block can trigger several others and it uses a very simple „join" unit for reuniting multiple operation paths. Concurrent execution becomes difficult, when concurrent paths try to use the same resource. The basic methodology offers a local, separate competition handling unit, a so called „competition manager" (CM). In a structured way, a competition manager is constructed of slices. One slice handles one competing block. CM slices are chained to form a CM unit. This unit can arbitrate any building block without any modification. Fig. 5 shows the block scheme and the state diagram of the CM slice. Depending on the number of the competing „resource managers" the same amount of competition manager slices have to be combined.

Access request is signalled to the competition manager via its „reqi" input. If priority situation allows, it forwards the signal through its „acko" output to the corresponding competing „resource manager". When the given resource manager finishes, its „rdy" output is fed back to the competition manager's „rdyi" input. (It's a branch, as „rdy" triggers the subsequent operation too.) Priority situation is arbitrated by a daisy-chain and

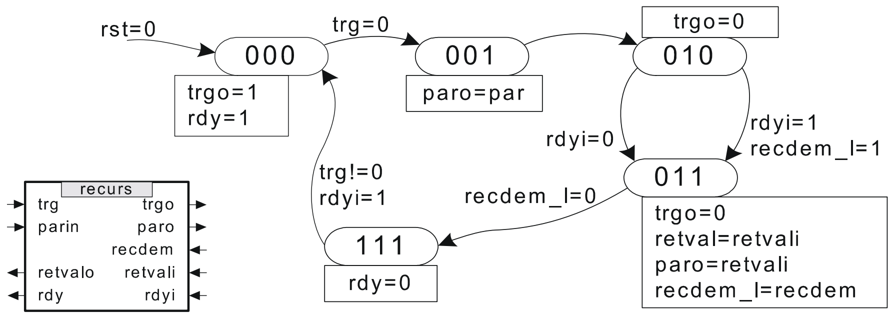


Fig. 3. Block scheme and state diagram of the "recurs" unit

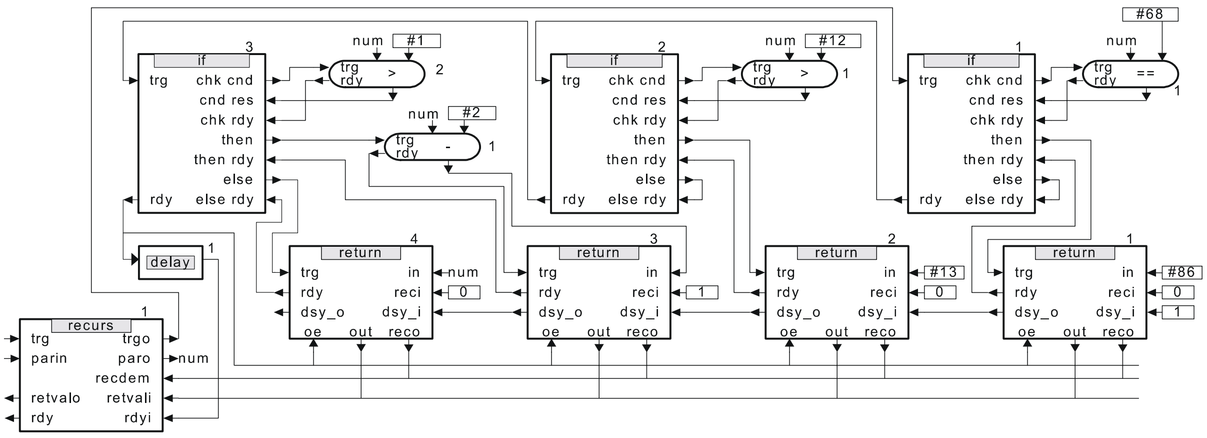
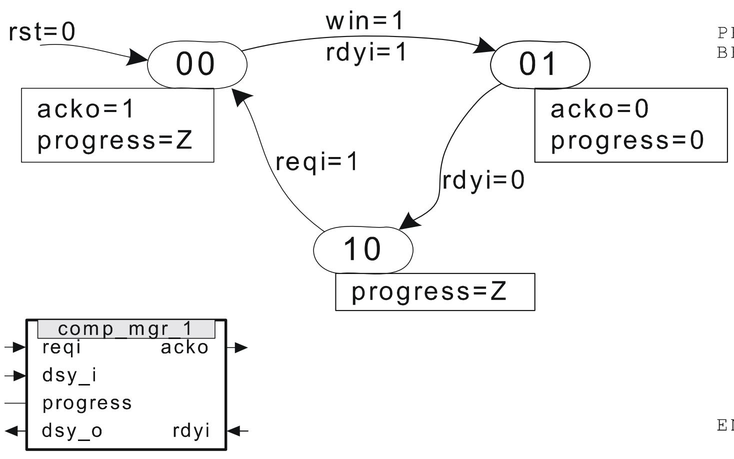


Fig. 4. Block scheme of func_r()



```
PROCESS (reqi, dsyi, progress, rst, status)
BEGIN
  IF status = "00" THEN
    IF reqi = '0' AND dsyi = '0' AND
       progress /= '0' AND rdyi = '1' THEN
      win  <= '1';
      dsyo <= '1';
    ELSE
      win  <= '0';
      dsyo <= dsyi;
    END IF;
  END IF;
  IF status = "01" THEN
    win  <= '1';
    dsyo <= '1';
  END IF;
  IF status = "10" OR rst = '0' THEN
    win  <= '0';
    dsyo <= dsyi;
  END IF;
END PROCESS;
```

Fig. 5. Block scheme and state diagram of the „competition manager"

a progress signal. The CM sets its „dsyo" output if its „dsyi" input is set, or if its „dsyi" input is not set and it has an active „reqi" input. The CM having an inactive „dsyi" input and a set „dsyo" output can activate its „acko" output. Avoiding situations when a higher priority request joins (or rather interrupts) a running competition and takes over the control, a progress bus is introduced connecting all CM slices. A CM slice can enter a competition only then, when the progress bus is not driven yet. If a slice enters, it drives the progress bus low.

Using these and other blocks, the following scheme represents the „main()" function on Fig. 6.

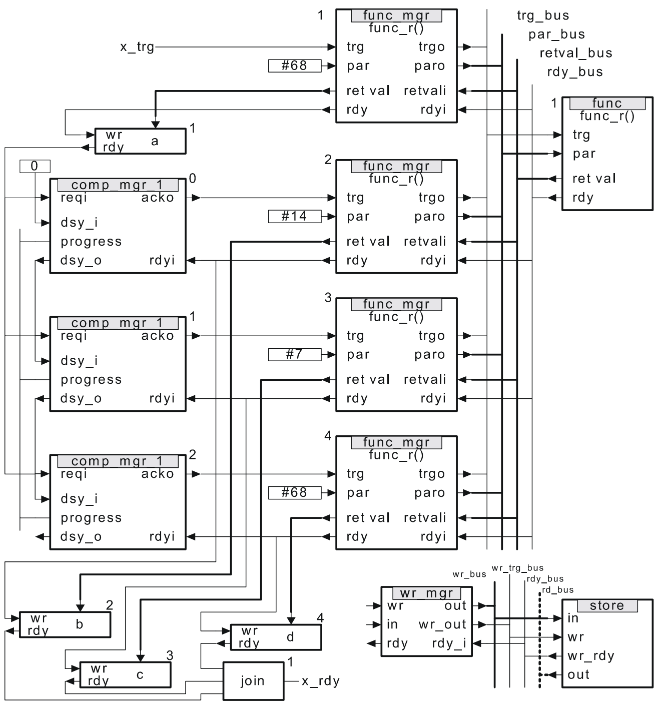


Fig. 6. Block scheme of the „main()" function

The „main()" function is started by „x_trg". It triggers the first function call, or rather a function manager (func_mgr #1) to "func_r()". It reaches the single sample of "func_r()" via a bus system consisting of "trg_bus", "par_bus", "retval_bus" and "rdy_bus". This time no competition is given as "func_mgr #1" is triggered alone and until it's done, no other manager of "func_r()" is triggered. "func_mgr #1" first drives the "par_bus" with the value put on its "par" input. In the next clock cycle it triggers "func_r()" via "trg_bus". When "func_r()" is done, it drives the "retval_bus" with the actual return value and in the next clock cycle it "back-triggers" the actual calling function manager (func_mgr#1) via the "rdy_bus". As "func_mgr #1" gets the "rdyi" signal, it undrives "trg_bus" and "par_bus" causing "func_r()" to undrive "retval_bus" and drive "rdy_bus" inactive.

As "func_mgr #1" is done, it triggers "wr_mgr #1", which is responsible to write variable "char a" stored in "char_store_a". This then triggers three competition manager slices at once ("comp_mgr_1 #0", "comp_mgr_1 #1", "comp_mgr_1 #2") building up "comp_mgr_3 #1", which is a 3-channel competition manager. The slices are daisy-chained and - in addition – are tied to a common bus called "progress". The competition manager is necessary, because "func_r()" is to be called three times concurrently, but - in this example - it has only one single sample. These competition manager slices arbitrate access to "func_r()" via the same bus system as "func_mgr #1" did. When a function manager is done, it signals its readiness via its "rdy" output. These outputs trigger each respective character write manager, writing char b, c and d.

The way as a variable storage is accessed by a corresponding write manager is very similar to the one as a function sample is accessed. At the lower right "wr_mgr #1" is drawn in detail together with the corresponding storage unit ("char_store_a"). All variables have such an arrangement, but for the sake of simplicity all write managers are drawn simplified and corresponding storages do not appear in this figure.

Timing diagrams of „main()" can be seen on Fig. 7.

The timing diagram of Fig. 7 is sectioned into six sections. One section has been omitted containing reset, clock, main trigger and main ready. These will be included and discussed later.

The first section (char a,b,c,d) contains the write and read buses of these variables. It can be seen that at first a value appears on the write bus, then – according to signals not shown here - the read bus takes it over.

In the next section (func_r interface) signals interfacing „func_r()" to „main()" appear. Here at first #68 (0x44) is put on the „par_bus", then „trg_bus", with an active low triggers „func_r()". When the function is ready, it puts the return value #86 (0x56) onto „retval_bus", then signals its readiness on „rdy_bus" with an active low. Signals „recdem" (recursion demand) and „recdem_l" (the latch in „recurs" storing the value „recdem") show that this return does not need recursion. In the first section it now can be seen as „char_a" gets its new value. The next call is done with a parameter value of #14 (0x0E) for which a return value of #13 (0x0D) is given, which then is written into „char_b". The next call is done with a value of #7 (0x07) which forces „func_r" to recursion. This can be observed on the „recdem" signals just like on the „retval_bus" which – for diagnostic purposes – shows the actual „num" value (see Fig. 4.) but - without „rdy_bus" activation - control remains at „func_r". When the recursion has reached the exit condition (num<=1), „func_r" quits and the real return value #1 (0x01) is put on the „retval_bus". This value is then written into „char_c" (see first section). Then „func_r" is called with #68 again.

The next section shows signals of function „main()" from which „progress" is to be observed when the competition managers (last three sections) are discussed. Based on Fig. 5 the followings have to be noted here. There are three function managers triggered concurrently, which then have to be arbitrated via connected competition manager slices. It can be seen in Fig.7. as three „reqi" signals get active at the same time (at 900nsec). The daisy chained competition manager slices (comp_mgr_1_x x=0,1,2) start arbitrating the situation. By the

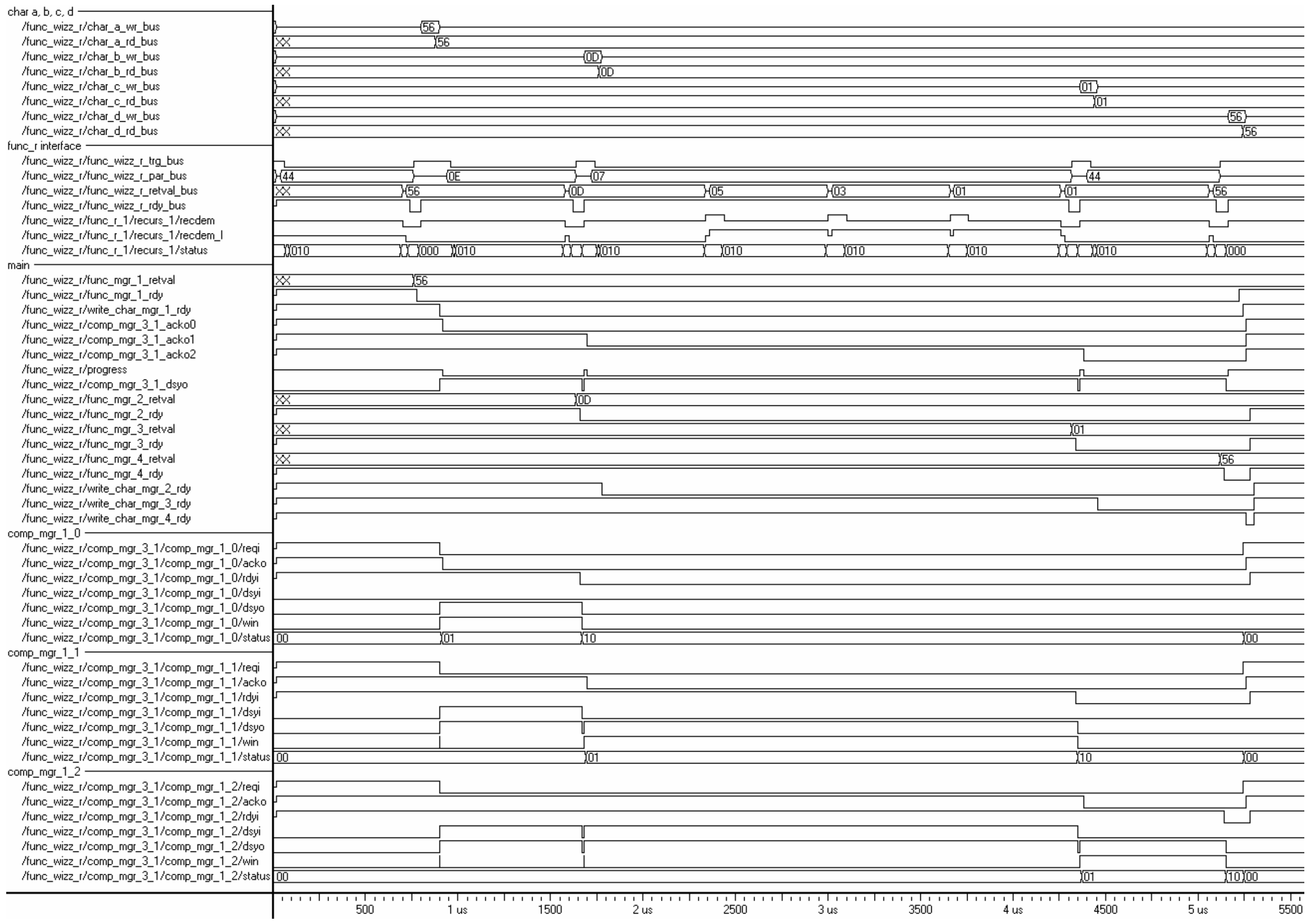


Fig. 7. Timing diagram of the application of Fig. 6

priority chain set up by „dsy_i" and „dsy_o" signals the priority is determined by an asynchronous evaluation within a half clock period. The competition manager slice is the winner in which the internal signal „win" can get an active high value. This drives the „progress" bus low. Then the corresponding „acko" (acknowledge out) signal gets active and the corresponding function manager is triggered. When it is done, its „rdy" output signals to the corresponding competition manager slice on its „rdy_i" input. Then this slice releases the „progress" bus and ends competing.

Pipelining

The application shown in the previous section was an example for „horizontal" paralleling, when independent operations are run concurrently. In case of strong sequentialism - like chain calculations -, this parallel triggering is not possible. But, if such calculations are parts of a cycle, "vertical" paralleling, or pipelining is the way out.

Although during the execution of a desktop computer SW - originally written in C - many pipelines are working, these work on machine code level - within the CPU – and in the source code there is no instrument to achieve or to control it. In addition, C can not cope with "time-overlapping", when at different pipeline stages, values of different times of the very same variable are needed. Finally, handling of unbalanced or even data-dependent execution times of the individual pipeline stages is also not solved.

Some C-based hardware definition methods [2,6,12,13,14,15] have instruments for pipelining, however here the developer has to prescribe this function by a specific instruction to which a special variable type can belong, or needs special preconditions, instruments or has to construct pipelines manually.

This section shows that a simple and effective method has been found that fits the transparent and flexible nature of the embedding methodology. No special keyword, instrument or wrapping is needed, not even special variable types. Time-overlapping of different values of the same variable is solved, just as the use of any variable types. In addition, a solution has been found to cope with different duration pipeline sections including when duration is data dependent. This costs only one single clock cycle per pipeline section and a small state-machine. Automatic compile-time code analysis – as mentioned earlier - can make the decision too, whether pipelining makes sense and if so, how the pipeline should be sectioned (see subsection Applicability).

It has been found that any cycle instruction of the C language can be used for pipelining; only the cycle body has to be transformed to a "cycle-and-tail" arrangement. Fig. 8 shows how a

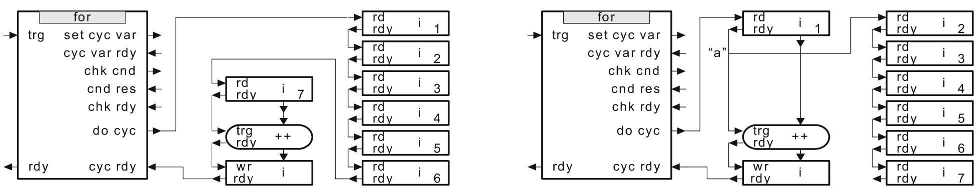


Fig. 8. Cycle-and-tail arrangement

"for" cycle is converted by this manner.

As shown in Fig. 8, instead of having block 1..6 in the cycle body (see left), only block 1 is there, while the rest just hang on it as a tail (see right). The „rdy" output of read block 1 is branched at „a", so the state machine of the „for" instruction believes that the cycle body is done, while execution of the „tail" blocks goes on. After the „for" has positively checked, that the cycle body has to be executed again, it triggers the operations of the cycle body. This means, that there are two points of execution, one in the cycle body and one in the tail. The number of points of execution proceeding towards tail-end depends on the length (duration) of the tail and of the cycle. When a pipelined cycle instruction finishes - depending on the length of the tail - corresponding time has to ellapse to let the tail finish completely and respective data settle.

However the above basic idea is simple, several problems remain that are addressed and solved in the next subsections.

Left-alone function of blocks

Function of blocks in the tail are not controlled anymore, so these must be able to function even when they are „left alone". While the embedding methodology ensures, that a chain of blocks (like the cycle body of a „for") will not be restarted until the final block has given a ready, in pipelined mode several restarts are possible before the first ready arrives at the end. Consequently, each block has to go on functioning when once triggered even if „trg" gets inactive meanwhile. When a block is done, it has to activate its „rdy" output for at least one clock period even if the trigger input is not active already. This is called here „left alone" function, meaning that once triggered, the block will be certainly executed, even if the trigger input isn't active anymore. Retriggering of a working block (e.g. instruction unit, operation unit) still must be avoided.

Data consistency in the pipeline

On Fig. 9 a schematic of a simple application and its timing diagrams can be seen. It illustrates well the data consistency problem.

Fig. 9 shows how a data consistency problem occures, when the tail is long enough to overlap with the new start of the cycle body. For the sake of siplicity this application is only a simple „for" cycle, where the cycle body is only a read-increment-write of the cycle variable and the tail consists of 11 read blocks, reading the value stored in „char_store_i".

At „A" the storage of „char i" gets a new input value and at „B" the write is triggered. At „C" the new value appears on the corresponding read bus and at „D" the storage gives a ready signal. At „E" – after a similar sequence – the next „i" value gets to the read bus.

Note that while the first four read managers (from „F" to „G") read the actual value of „i", the rest – due to their timely position – read the next „i" value (from „H" on).

This test shows that a mechanism has to be created, which serves operations in the pipeline with corresponding data. For this mechanism the „data spreading" technique is proposed, where input and output data travel with the corresponding point-of-execution, so even if data changes in the corresponding storage, operations get the specific older value, what they have to use.

The goal - not to introduce demanding specific instruments for pipelining [2] - can be kept with the special use of the existing „read" operation. A „read" block has an input port, which is normally connected to the „read_bus" of the corresponding variable. When the „read" is triggered by its „trg" input, it latches and outputs the value of the „read_bus". This output is held even after „trg" gets inactive and so „rdy" gets inactive too.

It has been figured out that if such „read" blocks are not fed by the original variable storage, but by each other, then - with appropriate triggering - values of given variables can "spread with" the operations and data consistency can be kept. Fig. **??**10. shows how data spreading "read" blocks have to be placed and triggered among operation blocks. In Fig. 10 a part of a cycle-

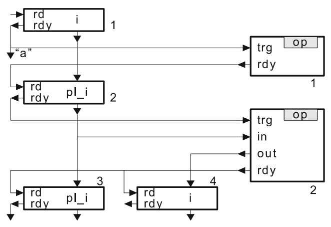


Fig. 10. Data spreading. Placement and triggering

and-tail arrangement can be seen. Block "read #1" is in the "cycle", but the rest of the blocks belong to the tail. So the "rdy" output of "read #1" is branched and path "a" triggers subsequent "cycle" blocks, while the other path triggers "op #1". When "op #1" gets done, it triggers "read #2", which is a data spreading "read". The specialty of "read #2" is that it does not read the read bus of variable "i", but it reads the output of "read #1", so even if variable "i" has changed meanwhile in its storage, a previous value is handed over to "read #2".

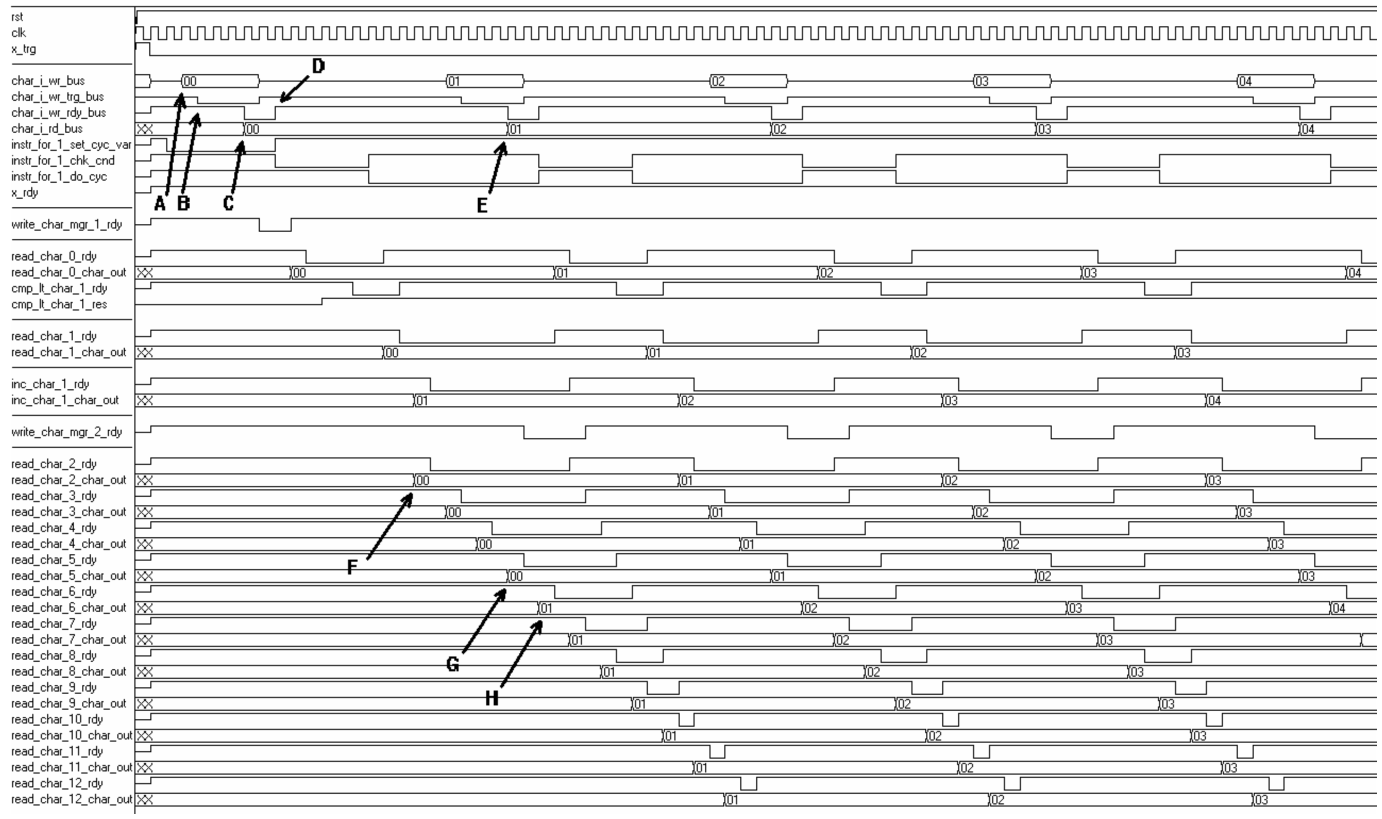


Fig. 9. Data consistency problem during pipelining

Run-up in the pipeline

As "left alone" function and "data spreading" is solved, there is still a problem, arising from the execution time difference of blocks in the tail. Introducing delay blocks to balance the tail could be a bulk solution, but in case of data dependent execution time, it won't do either.

Fig. 11 shows an illustration, how different execution time blocks hinder pipelining

In Fig. 11 a "for" block can be seen with a cycle-and-tail arrangement and the corresponding timing diagram. It is not drawn on the figure, but let's consider a "for (i=0; i<3; i++) ... " program part. By the figure, "for #1" – after setting the cycle variable and checking the condition – triggers "op #1", which is the only operation of the cycle. The "rdy" output of "op #1" is split at point "a". One path goes back to "for #1", while the other triggers the first operation – "op #2" – of the tail. The operations have different durations. Operations #1, #2 and #4 are 3 time quanta, "op #3" is 11 time quanta and the time that "for #1" needs to re-check the condition and activate its "do cyc" output needs 2 time quanta. The timing diagram of the same figure has 3 lanes, representing how operations are triggered with consecutive "i" values.

At "t0" "for #1" is triggered on its "trg" input. By "t1" it sets the cycle variable and checks the condition, so at "t1" operation #1 is started. As "op #1" is done at "t2", it triggers both "for #1 / cyc_rdy" and "op #2" so from this time two lanes are needed to describe, what is happening. At "t3" "op #3" is started while "op #1" (with i=1 already) is running. After the second run of "op #1", at "t4", a third lane is necessary to represent operations regarding i=2. By "t5" a situation occurs, which can not be resolved: "op #3" is still running with i=0, but is already needed to run also with i=1. By "t6" the situation is even worse, where three examples of "op #3" should ensure proper function. As an operation can have one single status at a time, the arrangement on Fig. 11 is not feasible.

As a consequence, a mechanism is needed, that can cope with different operation length in the tail.

The solution found is a block, which can pause triggering the next operation block until that is done. This is called "pipeguard" and its task is to build up a feedback chain for operation pausing in case of pipeline jam. Fig. 12 shows the state diagram of the "pipeguard" and its basic application.

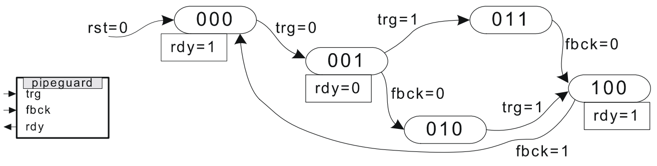


Fig. 12. Block scheme and state diagram of the pipeguard block

In Fig. 12a "pipeguard" block can be seen. It is a dynamic-duration block sensing the readiness of the subsequent block via its "fbck" input. It tracks the function of the next block and if that has been triggered once then it doesn't let that be re-triggered until that is done. In the state diagram, path 000-001-011-100 means a slower subsequent block than the previous, while path 000-001-010-100 means a faster subsequent block

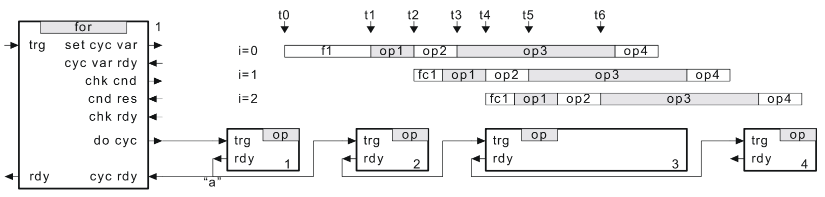


Fig. 11. Different execution time blocks in the tail and timing behaviour

than the previous. The block is constructed in a way that if the subsequent operation is free, than the "pipeguard" takes only one single clock period. The "pipeguard" can determine if the next block has been triggered, but its "rdy" output is inactive. Then it waits until the ready gets active and then inactive again.

Fig. 13 shows a simple example of using "pipeguards", extended with a pipeline timing diagram.

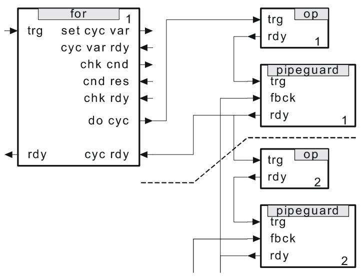


Fig. 13. Using pipeguards

In Fig. 13 an application of "pipeguard" blocks can be seen. It is a "for" statement in cycle-and-tail arrangement. The cycle consists of "op #1" and a "pipeguard" called "pg #1", while the tail is not fully drawn, but one stage with "op #2" and "pg #2" is presented. (See dashed line)

When "for #1 / do_cyc" triggers the cycle body, "op #1" starts and when it is done, it triggers "pg #1". This "pipeguard" first checks whether "pg #2" is done. If not, then "pg #1" waits and gives no ready signal. When the observed block is done, "pg #1" activates its ready output and so triggers both the "for #1 / cyc_rdy" and "op #2 / trg". Here "op #2" can start and then execution is halted at "pg #2" if necessary.

Chaining "pipeguards" backwards with their "fbck" and "rdy" connections solves pipeline jamming problems for any tail length and any timing extremes of tail blocks.

Complete proposal

An example application is shown on Fig. 14 regarding the complete proposal for pipelined execution.

A short software module (as included in the upper part of Fig. 14) is translated by the methodology. The application is a "for" cycle adding two ten-element-arrays building the result in a third array. Data dependent execution time is caused by an "if" statement, which - in case of the 2^{nd} element - rewrites the result to 2. A "delay" is added in the resulting block scheme to increase the time spent with the taken "if". Such a SW in pipelined execution requires all techniques introduced above in this paper.

Function of the resulting C-based hardware starts with "for #1", which first sets the cycle variable to zero ("wr #1", variable storages are not shown on the drawing), then checks the condition using "rd #1" and "cmp_lt #1". The cycle body of "for #1" is represented in a cycle-and-tail arrangement. The "cycle" part consists of "rd #2", "pg #1", "inc #1", "wr #2" and – not to forget – the part of "for #1" dealing with re-checking the cycle condition. The "cycle" is tapped after "pg #1", which is right after "rd #2" (see branch point "a"). This makes the earliest start of the "tail" possible. The "pipeguard" in the "cycle" watches the "tail" to avoid block re-triggering in the whole "tail". When "pg #1" triggers the "tail", first it triggers a data spreading read ("rd #3") to assure the right "i" value spreading with pipelined operations ("pl: i" stands for pipelined "i"). In parallel, the first pipeguard triggers "inc #1" too, which then increments the cycle variable. (Dashed lines emphasize pipeguarded pipeline section boundaries.)

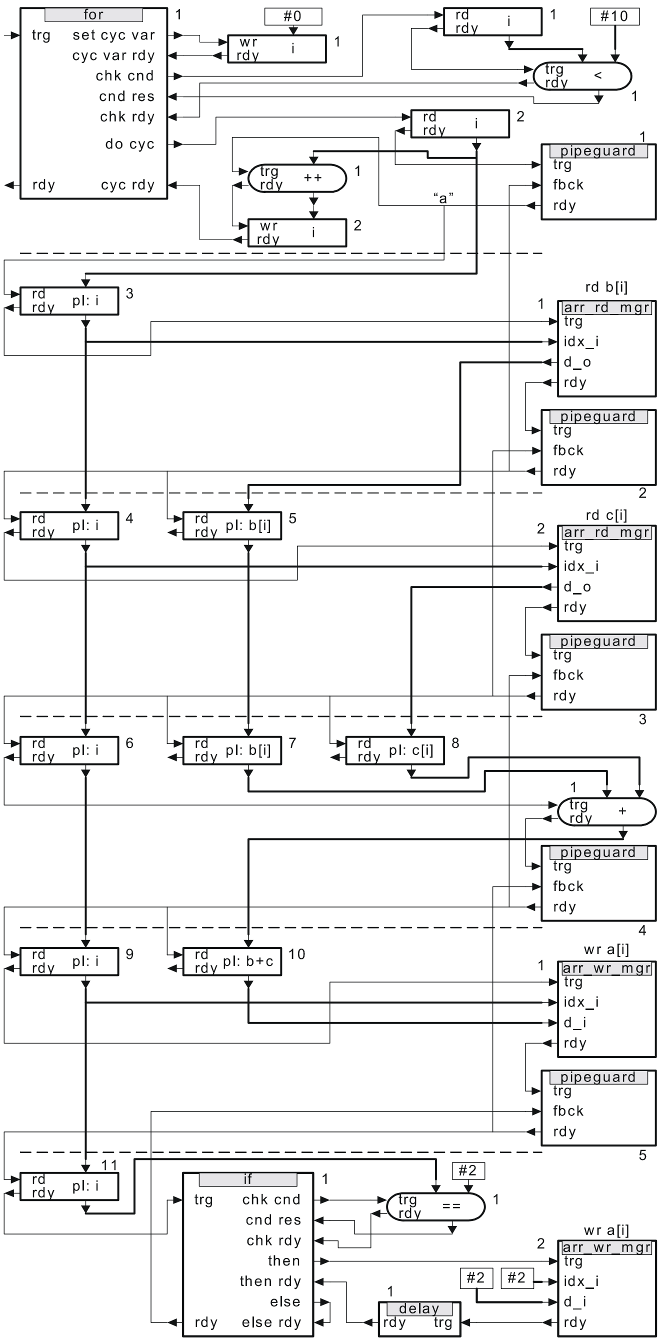
After "rd #3" reading b[i] is done by "arr_rd_mgr #1" another "pipeguard" ("pg #2") is triggered. When pipeline situation allows, "pg #2" starts the next pipeline section by triggering "rd #4" and "rd #5" in parallel. These are data spreading reads too. Read #4 is to spread on variable "i", while "rd #5" is to start spreading the corresponding value of "b[i]".

The same happens in the next pipeline section with "c[i]", but then "rd #6", "rd #7", "rd #8" are needed to spread the consistent values of "i", "b[i]" and "c[i]". After these three reads, "b[i]" and "c[i]" gets added and the value of "b[i]+c[i]" is spreaded on together with "i". In the next section the addition result is written by "arr_wr_mgr #1" using the index value provided by "rd #9".

The last section of the "tail" is a data dependent execution time "if" statement. If "i" equals #2, then "a[i]" is rewritten to #2. (A delay block is added here to make timing diagrams during evaluation better understandable.) Here – at the end of the pipeline - no "pipeguard" is needed, so the last "rdy" signal of this section is fed back to the previous "pipeguard".

Note that thank to the back-chaining of the "pipeguards" via their "rdy" and "fbck" connections, a delaying execution of any pipeline section leads to the halt of previous sections. As soon as the long running section gets free, the pipeline continues execution in all sections.

The right side of Fig. 14 shows the pipeline timing. Each vertical lane represents an execution sequence for a given "i" value. Due to the different and changing execution times of the blocks and pipeline sections, only the entry point of each lane is



```
for (i=0;i<10;i++)
    {
    a[i] = b[i] + c[i];
    if ( i == 2 ) a[i] = 2;
    }
```

i = 0

```
f1/trg
wr1
f1/chk
rd1
cmp_lt1
f1/do_c
rd2
pg1
rd3
ard1
pg2
rd4,5
ard2
pg3
rd6,7,8
add1
pg4
rd9,10
awr1
pg5
rd11
if1/trg
cmp_eq1
if1/chk
if1/else
```

i = 1

```
inc1
wr2
f1/cycr
rd2
pg1
rd3
ard1
pg2
rd4,5
ard2
pg3
rd6,7,8
add1
pg4
rd9,10
awr1
pg5
rd11
if1/trg
cmp_eq1
if1/chk
if1/else
```

i = 2

```
inc1
wr2
f1/cycr
rd2
pg1
rd3
ard1
pg2
rd4,5
ard2
pg3
rd6,7,8
add1
pg4
rd9,10
awr1
pg5
rd11
if1/trg
cmp_eq1
if1/chk
if1/then
awr2
dly1
if1/rdy
```

i = 3

```
inc1
wr2
f1/cycr
rd2
pg1
rd3
ard1
pg2
rd4,5
ard2
pg3
rd6,7,8
add1
pg4
rd9,10
awr1
pg5
rd11
if1/trg
cmp_eq1
if1/chk
if1/else
```

Fig. 14. Application example with pipeguards

given, operation overlapping of different lanes is changing.

Fig. 15 shows important timing behaviour of the application of Fig. 14.

In Fig. 15 relevant timing diagrams of the shown application can be seen. The uppermost section contains basic signals like low active reset ("rst"), application trigger ("x_trg"), the single phase clock ("clk") and the final ready signal called "x_rst". The next section ("char i") contains signals for variable "i". First a value is transmitted to the store on "char_i_wr_bus", then the write process is triggered via a negative edge on "char_i_wr_trg_bus" (however it looks like a Z->L transition, it is a H->L one where H is maintained by a pull-up circuit on the bus). When the write process is done, the store signals its readiness via "char_i_wr_rdy_bus" and the value written appears on "char_i_rd_bus" where from any read operation can access it.

The next section ("array a") contains signals for array variable "a[]". The index value is not presented here. Here an access request signal ("a_req_bus") and a read/write signal ("a_rdwr_bus") is used. First the index bus and the rd/wr bus is set, then the "req" signal goes low to initiate the access at the array store. When the appropriate value is put out by the array store to "a_data_bus", it signals on the "a_rdy_bus" with a low. (Here no read bus is used, as an array store can output only one value, so the actual value is handed over on the data bus to the active array manager unit.) Sections "array b" and "array c" have the same behaviour.

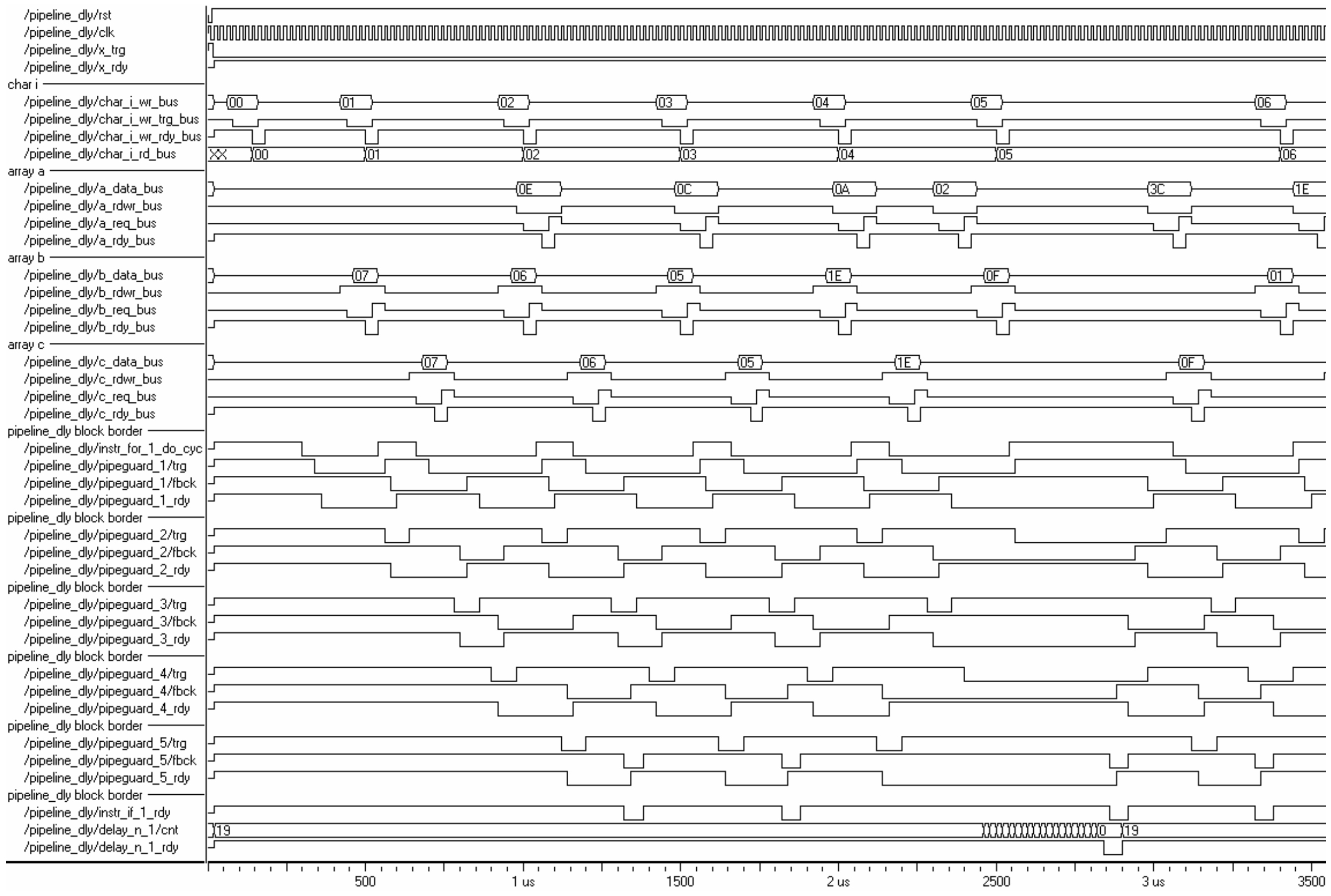


Fig. 15. Timing behaviour

The next sections are divided by "block borders", where almost only signals of the different "pipeguards" are included, which are sufficient to show the main feature of the function.

First "instr_for_1_do_cyc" starts the cycle body. In a few clock periods "pg #1" is triggered. As "pg #1" "knows" that the next section is ready to be triggered, it triggers it after one clock cycle. This goes on with all sections of the "tail" with i=0. In the next round – with i=1 – however "pipeguard_1_rdy" does not follow "instr_if_1_do_cyc" as before, because subsequent sections force a delay via the "fbck-rdy" chain. This effect can be observed in several samples on the diagram.

Another short introduction is necessary regarding the variable section to follow corresponding values. On "char_i_rd_bus" values of 00, 02, 02, 03, … can be seen as "for #1" proceeds. Corresponding values of "b_data_bus" and "c_data_bus" are 07, 06, 05, 1E, … similar prestored values. Sum values being written to array a[] are 0E, 0C, 0A, 02 and 3C on "a_data_bus". The reason why five values are mentioned is that a[2] is rewritten from 0A to 02 because of the taken "if" in case of i=2. Characteristic to the pipelined function, at e.g. 960nsec variable "i" is being written to "02", but this value does not yet appear on its read bus; a[0] still has not got its calculated value defined by b[0] and c[0], however b[] is already being read for i=1.

As a[2] has been written to 0A – due to the taken "if" – it is rewritten to 02 (at 2300nsec). In addition, "delay #1" is started (realising a preset 20 clock cycle delay). This delay only helps understanding the way "pipeguards" are functioning. Between 2150nsec and 2920nsec the whole pipeline comes to a halt, caused by the delaying "if" at the end of the line. When "delay #1" gets done, its "if" gets done too (at 2920nsec) and the whole pipeline goes on working again.

So, the basic idea of creating a "cycle-and-tail arrangement" from the cycle body of any cycle instruction needs some extensions. These are "left-alone" function ability of the blocks, "data spreading" to maintain data consistency in the pipeline and using "pipeguard" blocks against run-ups in the pipeline.

Applicability

Applicability of this pipelining method is very wide. Possibilities are limited by economical boundaries. A strong economical boundary is, when setting up a pipeline has no time benefit, but costs extra apparatus. Such are operation blocks that do not run repeatedly, but only once (e.g. start-up initialisations); or a cycle with purely independent operations that better run parallel than in pipeline. (Mostly a cycle body consists of dependent and independent operations. So pipelining groups of paralleled operations may give the best performance.) Operation blocks that are not part of a cycle may also need to be pipelined if these run repeatedly. Such case is, when a SW-HW codesign's HW part is defined in C as a computation sequence and is "called" repeatedly by a SW part.

When pipelining is chosen, the compile-time analysis tool faces an NP-hard task. Here speed and cost constraints have to be fulfilled by the setup and sectioning of the pipeline. The smaller the pipeline sections are, the shorter restarting period is provided. However the more pipeline sections created, the more it costs. The cost ramp is progressive because more variables have to be spread and for more sections.

When setting up a pipeline for a certain code part, it has to be analysed how the cost of competition management of shared resources relates to the cost gain caused by the resource sharing itself. Consider a multiplier that is shared among more operations in a cycle body. Till these run sequential, the multiplier can be shared without any problem. As soon as this cycle body is run in pipeline mode, the operations may need the single multiplier concurrently. This situation can be handled by a competition manager as proposed, but it has extra costs.

Conclusion

The transparent and structured C-to-hardware translation base methodology published earlier by the authors has easily embedded an add-on methodology enabling pipeline fuction of cyclic operations. So, „horizontal paralleling" of the embedding methodology has been extended by a „vertical paralleling" of the add-on methodology. It has very low and proportional overhead regarding enabling apparatus and only one single clock cycle per pipeline section regarding operation time. The enabling apparatus consists of a simple, state machine based „pipeguard", a specific reuse of an already existing building block of the base methodology and a specific handshaking technique of the building blocks.

Two example applications highlight both horizontal and vertical paralleling techniques and simulator results prove the correctness and effectiveness of the whole methodology.

Further research

However pipelining can be solved with the add-on methodology discussed above, several questions remain open regarding the way it is used. The most important task is the pipeline sectioning, having major influence on speed and economy. In addition, extremes have to be analysed in pipeline applications to provide statistic data regarding execution time and resource optimal setups.

Determining cycle-tail border and sectioning the pipeline (the tail) is an optimisation question. Each section needs a "pipeguard" and so many data spreading read units as many values have to be propagated. A "pipeguard" needs one single clock cycle, so usually it is only a small fraction of the whole section time. First the cycle-tail border has to be chosen. As the longest running pipeline section determines the pipeline execution time, main considerations have to aim at a time-balanced pipeline. So the border has to be determined so, that the cycle's average execution time is close to the foreseen average pipeline section execution times in the tail. This choice has a decisive influence on the granularity of the pipeline. A long running "cycle" determines a "tail" with less, but longer running sections. Then the "tail" has to be cut into sections so, that average execution times of the sections are closest to even. Extremes of pipeline sectioning spread from no-tail, only cycle to one, short running block in the cycle and block-by-block sectioned tail with short running blocks. A restarting period of one single clock cycle can not be reached, as a working block must not be retriggered and even a short block takes a few clock cycles. Still a very high throughput can be reached.

This pipelining enabling methodology has no theoretical limits however a pipeline tail with several sections and with big amount of spreaded variables can consumpt considerable resources.

References

1 **Fujita M, Nakamura H**, *The Standard SpecC Language*. CAPSL, BoF Session at SC2002, Baltimore.

2 **Dömer R, Gerstlauer A, Gajski D**, *SpecC Language Reference Manual Version 2.0*, 2002. SpecC Technology Open Consortium.

3 **Gerstlauer A, Dömer R, Peng J, Gajski D**, *System Design – A Practical Guide with SpecC*, Kluwer, May 2001. ISBN 0792373871.

4 **Bowen M**, *Handel-C Language Reference Manual Ver 2.1*. Embedded Solutions limited.

5 **Baird M**, *System-C 2.0.1 Language Reference Manual Rev 1.0*. Open SystemC Initiative , 1177 Braham Lane 302, San Jose, CA 95118 – 3799.

6 **Self RP, Fleury M, Downton AC**, *A Design Methodology for Construction of Asynchronous Pipelines with Handel-C*, IEE Proceedings Software 2003 **150** (2003), no. 1, 39-47, DOI 10.1049/ip-sen:20030206.

7 **Arató P, Csák B, Kandár T, Mohr Z**, *Some components of a new methodology of system-level synthesis*, INES2002 Hotel Adriatic (Opatija, Croatia), May 2002.

8 **Arató P, Csák B**, *Hardware Software Co-Design Based On Standard C-Language Source code*, ICCC 2003 August 29-31 2003.

9 ______, *Programming Language Based Definition of Application Oriented Hardware*, WISP 2003 September 4-6 (Budapest, Hungary, 2003), DOI 10.1109/ISP.2003.1275836, (to appear in print).

10 ______, *Hardware Definition Based On Standard C-language Source Code*, FDL '03, September 2003.

11 ______, *Solutions for Competition Cases in C-Language Defined Application Specific Hardware*, ICCC04, 2004 IEEE International Conference on Computational Cybernetics, 30 August, DOI 10.1109/ICCCYB.2004.1437666, (to appear in print).

12 **Ziegler H, So B, Hall M, Diniz P**, *Coarse-Grain Pipelining on Multiple FPGA Architectures*, Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, DOI 10.1109/FPGA.2002.1106663, (to appear in print). FCCM2002, Napa, California, 21-24 April, 2002.

13 **Gill G, Hansen J, Singh M**, *Loop Pipelining for High-Throughput Stream Computation Using Self-Timed Rings*, ICCAD 2006. IEEE/ACM International Conference, Nov 2006, DOI 10.1109/ICCAD.2006.320135, (to appear in print).

14 **Maruyama T, Hoshino T**, *A C to HDL compiler for pipeline processing on FPGAs*, FCCM2000, 8th IEEE Symposium on Field-Programmable Custom Computing Machines, 2000 17 April, DOI 10.1109/FPGA.2000.903397, (to appear in print).

15 **Rodrigues R, Cardoso J**, *On Pipelining Sequences of Data-Dependent Loops*, Journal of Universal Computer Science **13** (2007), no. 3, 419-439.