OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

VHDL Analysis of Pipelining Capabilities across Von-Neumann, Harvard and FPGA systems

Seminararbeit
from

Reilly Ertman
Err35230

OTH Regensburg
Faculty Informatics and Mathematics

Grading Professor: Prof. Dr. Rudolf Hackenberg

28. July 2023

Inhaltsverzeichnis

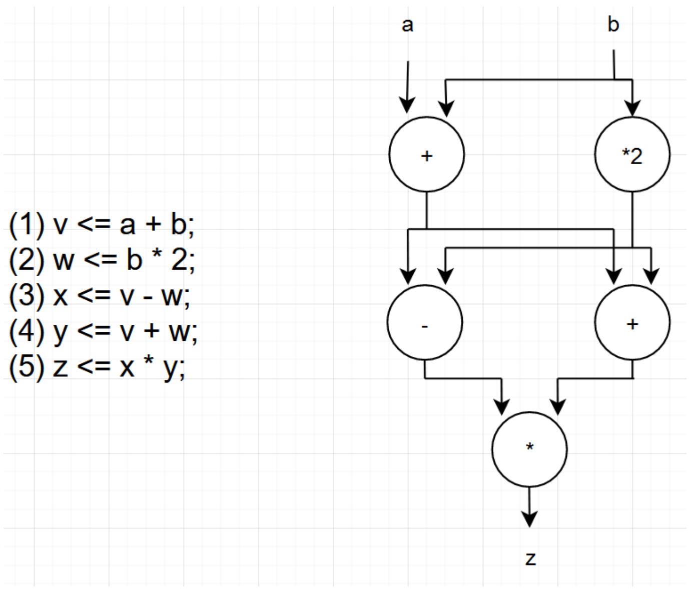
1 Introduction

The goal of every processing system is speed. According to Moore's law, the number of transistors on an integrated circuit doubles regularly. While this has held true since the 1970s, electronic engineers are reaching the physical boundaries of what is possible with hardware. Transistors, the building block of an integrated circuit, has real physical limits. The constant upward trend of Moore's law is slowly flattening. Therefore, in recent decades, the goal posts of an ideal processing system have been moving away from densely packing transistors to achieve fast speeds and towards improving efficiency via pipelining.

When computer science students sit down in the classroom for their first lecture, they are taught that a computer's CPU is a single sequential, blocking system. When the CPU picks up a job, it is busy until that job completes. However, for modern CPUs, this is no longer the case. Modern CPUs employ a strategy called pipelining, wherein they can perform several tasks *p*seudo-simultaneously, thereby increasing job-output significantly.

In this paper, I will analyse the fives lines of instruction in figure 1 and their corresponding ideal dataflow in detail:

I will model the code's sequential execution in both Von-Neumann and Harvard CPU



```
(1) v <= a + b;
(2) w <= b * 2;
(3) x <= v - w;
(4) y <= v + w;
(5) z <= x * y;
```

Abbildung 1: Sequential and parallel data flow

architectures using VHDL to construct and Modelsim to simulate their behavior and compare their potential for pipelining. Furthermore, I will also make recommendations how to improve pipeline capabilities on both Von-Neumann and Harvard CPU systems.

1.1 Sequential and Parallel Processing

Before examining these architecture structures, we must review flow control rules of digital logic, as I use valid_in / valid_out logic to issue control over modules and components in my implementations. These flow control rules will aid us later to improve pipelining.

2 Digital Signal Control Flow

I intend to use VHDL code as an aid to describe the inter-working of a CPU. Therefore, a firm understanding of control logic is critical to designing a system with an array of moving parts that must work in unison. A processing unit is composed of many hardware and software components, which all fight for access to a finite amount of resources. To govern the operation of hardware, every data processing system requires an exterior layer of control logic.

Let's say, for example, the ALU is about to perform multiplication on two operands: Once the CPU leaves the decode phase and enters the execution phase, the ALU needs assurances that the data it reads from its input registers are the correct ones. If no system for assurances is in place, it may action on incorrect data. This not only waists the several clock cycles of computation, but its incorrect output will be fed into other processes. In figure one, the result of w <= a + b is used later as an operand in an addition in line four (v + w). Then the result of that execution is used yet again in another computation later on. Again, the ALU needs assurance that the operands in its input registers are the correct values. The code in line 4, only states that an addition with two variables are to take place. We do not yet have guards in place to ensure that the *c*orrect values are subtracted.

2.1 Vld_in vld_out

This missing layer of control logic that allows a process (in this case, the ALU) to execute only once valid data is present is called vld_in and vld_out logic. To mark data as valid, we set a flag at the ALU's vld_in input. When the ALU sees this flag is set, it immediately begins execution, regardless of the value at its input. When finished, it outputs a '1' to its vld_out output, which then drives the valid_in input of the next phase in the instruction cycle. With each phase of the instruction cycle possessing their own vld_in and vld_out I/Os, they can be sure that the data in their respective input buffers is the correct data and thus begin executing. Any data that appears in a process's input buffer will be ignored without a simultaneous vld_in. Note that a vld_out is output for a single clock tick before returning to zero. This ensures that an operation is carried out just once. Valid_in and valid_out logic is a simple and elegant method to control when sub-processes begin and end their execution.

Figure 2 shows 4 signals, vld_in, data in, vld_out and data_out that belong to a single process. This process executes on data packet 0x40 when vld_in is raised and finishes

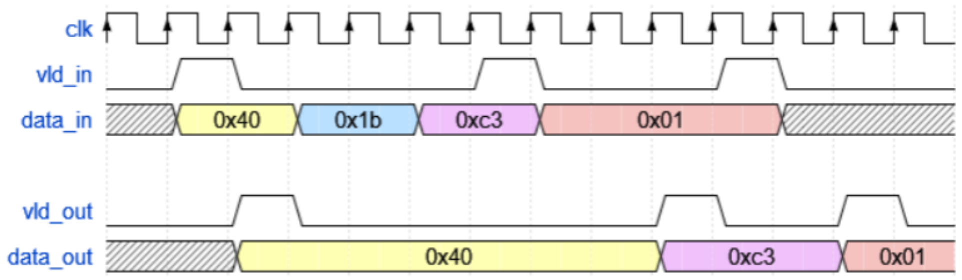


Abbildung 2: vld_in begins atomic process and vld_out signifies end.

execution several clock cycles later by raising vld_out for a single tick. Data packet 0x1b is ignored as the atomic process was busy executing. Once vld_out is set again to high, it accepts the data package 0xc3. With this gained understanding of control flow, we will better be able to understand how a single travels through a CPU.

2.2 Ready_out

In hardware systems, processes are executed atomically, meaning that once a process begins, it is busy with execution until finished. With these new guards in place, a process can only execute upon verified data. Yet we are still missing a crucial protection. In figure 2, the data packet 0x1b appears in the process's input buffer but is ignored as the process is busy executing. This presents an inherent issue: The data packet 0x1b may have been crucial data that was generated by the previous process in the chain, but it was sent while the next component was busy and thereby ignored.

Therefore, to avoid missing critical data, every component needs an additional single-bit output called rdy_out. If a process outputs a valid value, it should only be transmitted to the next component in the chain if that component is ready to accept it. If the next component is not ready, that device must also set its ready_to '0', so that it does not start a new job before passing on the value from its last job. Only once the next component is ready should the valid value be sent. Without checking if the next component is ready, this critical value will be lost.

Figure 3 exhibits this idea. In the area marked in red, the process receives data package 'A' and a vld_in simultaneously at its inputs. However, since its own rdy_out is '0', it cannot begin the job. Only in the next clock cycle, once it is finished with previous information does it see, accept and begin calculating the new data. Each time data is sent between processes, we must mark it as valid data and also check whether the next process is ready to accept it. Otherweise, that data must wait in a cache or the entire process must wait for the next component to be free.

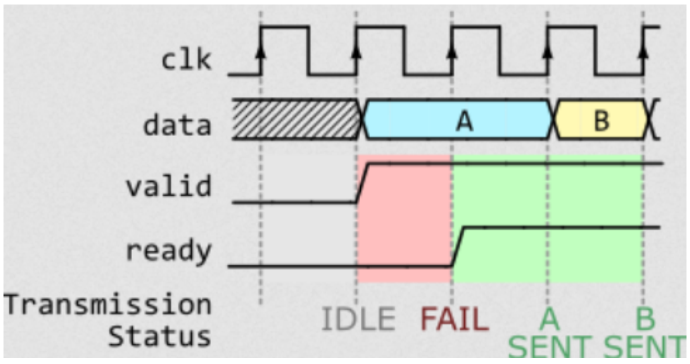


Abbildung 3: Data is only sent when valid and receiving module is ready

3 The Instruction Cycle

Between a user input request and the resulting output, data flows through the CPU, interacting with millions of transistors to complete the required computation. In my paper, the process of data traversing the CPU from beginning to end is broken down into 6 subprocesses: Instruction Fetch, Data Fetch, Decode, Execute, Store and Program Counter Incrementation.

3.1 Instruction Fetch

First, the job's instructions are retrieved from storage and placed in a local register with the following command.

```
sv_instruction_in      <= ram_data(ti_u(sv_prog_mem_addr));
```

As the program counter increments with each executed line of code, we use this counter value (sv_prog_mem_addr) to locate the next line of binary code in flash memory. This way, the next instruction address is bookmarked for quicker access.

In my simplified VHDL implementation of a CPU, an instruction is spread over 16 bits: The most significant four bits denote the mathematical or arithmetic operation to be carried out, called OP-Code. The next four bits detail the location in memory of the destination address, where the finished computation should be stored. The 8 least significant bits contain the location of my data variables in memory. Before passing the instruction to the CPU's ALU for computation, the CPU must know two things: (1) what logical or arithmetic operation should be employed upon the operands and also (2) the operands or variables themselves. Once we have fetched the instruction and stored it in the working register sv_instruction_in, the 4 bit instructions within the larger 16 bit instruction needs to be further fetch and decoded.

```
-- opcode 4bit(msb)//4bit destination//operand1 4bit//operand2 4bit(lsb)
sv_opcode              <= sv_instruction_in(15 downto 12);
sv_destination_addr    <= sv_instruction_in(11 downto 8);
sv_dereference_A       <= sv_instruction_in(7 downto 4);
sv_dereference_B       <= sv_instruction_in(3 downto 0);
sv_data_a_in           <= ram_data(ti_u(sv_dereference_A))(8-1 downto 0);
sv_data_b_in           <= ram_data(ti_u(sv_dereference_B))(8-1 downto 0);
```

3.2 Data Fetch

To further unravel the encoded instruction, we move to the phase Data Fetch. With the 16 bit instruction in a local register, we gather the required data operands by breaking them up into smaller pieces. If the data operands to be used are constants, then no further data must be fetched, but if the operands are variables, a further look-up is required to find the actual value. In the example of v <= a + b, the values operand1 and operand2 stored in sv_instruction _in are dereferenced and then used as inputs for the addition function.

3.3 Decode

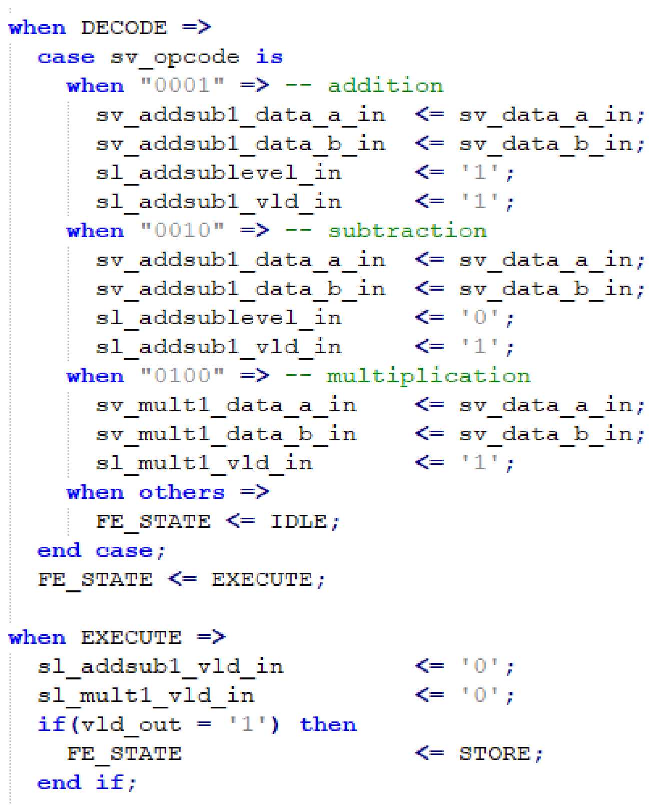
Once we have the data operands fetched, we must prepare the ALU for computation by accessing the needed arithmetic or logical operation and by sending the correct values to the ALU's inputs. Decoding the required operation is straight-forward: The decoder uses a simple look-up-table to determine the meaning of the OP-Code from amongst its toolbox of possible operations. In our example, the first command, v <= a + b, uses addition. We look at the 4 most significant bits of the instruction in sv_instruction_in, which is "0001", and compare "0001ägainst our toolbox of operations. Therefore we load the ALU responsible for addition with inputs.

3.4 Execution

As shown in figure 4, the CPU's ALU executes once its vld_in input receives the required flag. In this paper, I will treat the execution phase as a black box and not go into further detail, as this is not the focus of my topic. Once computation completes, the ALU outputs a vld_out, which the Store phase of the instruction cycle then sees as permission to begin storing.

3.5 Store and Program Counter

All that remains is to store the results back into memory and increase the program counter. For this, the calculated values are taken from the ALU's output register and written into RAM. To track its own location in code execution, the CPU increase the program counter, which will help deduce the next address of code to execute in the instruction phase.



```
when DECODE =>
  case sv_opcode is
    when "0001" => -- addition
      sv_addsub1_data_a_in  <= sv_data_a_in;
      sv_addsub1_data_b_in  <= sv_data_b_in;
      sl_addsublevel_in     <= '1';
      sl_addsub1_vld_in     <= '1';
    when "0010" => -- subtraction
      sv_addsub1_data_a_in  <= sv_data_a_in;
      sv_addsub1_data_b_in  <= sv_data_b_in;
      sl_addsublevel_in     <= '0';
      sl_addsub1_vld_in     <= '1';
    when "0100" => -- multiplication
      sv_mult1_data_a_in    <= sv_data_a_in;
      sv_mult1_data_b_in    <= sv_data_b_in;
      sl_mult1_vld_in       <= '1';
    when others =>
      FE_STATE <= IDLE;
  end case;
  FE_STATE <= EXECUTE;

when EXECUTE =>
  sl_addsub1_vld_in         <= '0';
  sl_mult1_vld_in           <= '0';
  if(vld_out = '1') then
    FE_STATE                <= STORE;
  end if;
```

Abbildung 4: Decode stage triggers execution phase

4 Pipelining

Consider a consumer good on a conveyor belt in a factory, where assembly requires 5 sequential steps. To complete the job, 5 workers are hired to assemble the product. When the first worker is done with step one, worker two continues with step two. At this point in time, worker one, three, four and five are waiting. Once worker two is finished with his task, worker three begins. Now workers one, two, four and five wait. Only once the product as been assembled does worker one work again. Assuming each task requires x minutes to complete, the output is x/5.

To increase throughput, the boss instructs worker one to begin assembling a second product once he gives the first task to the second worker. And once worker one finishes his task on the second product, he should begin step one on a third task, etc. With this change, the same group of workers now output 5 times more. This concept is called pipelining, which is illustrated in figure 5. At the first rising edge of the clock, Befehl 1 begins a fetch command and then executes a decode command thereafter, etc, until the write command. Then Befehl 1 terminates. *Pipelining* (2019)

Ideally, no process in the instruction cycles idles. The fetch command does not wait until Befehl 1 ends to execute again; instead, fetch is executed immediately after finishing a task, thereby increasing throughput of jobs significantly.

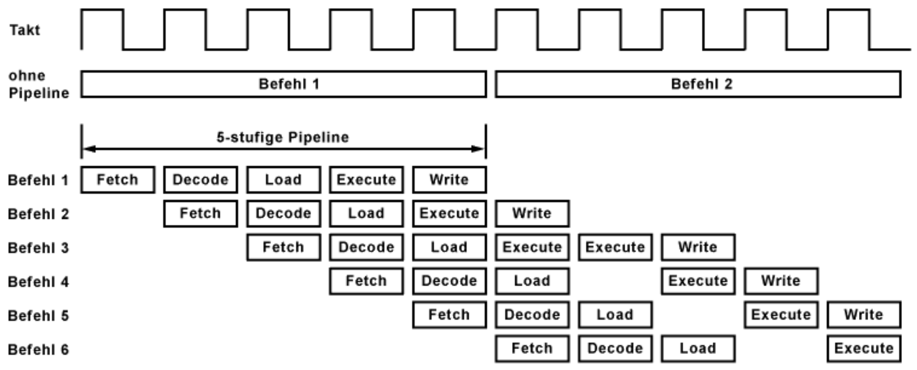


Abbildung 5: Nearly ideal pipelined instruction cycle

With our new understanding of digital logic control and the instruction cycle, let's look at an example of pipelining in VHDL. Figure 6 shows the five lines of our code being executed concurrently. The testbench inputs the values a = 7 and b = -5 and raises vld_in simultaneously to begin computation of code lines (1) and (2). The values v and w are calculated concurrently so that the values v = 2 and w = 10 appear simultaneously at the ALU's respective outputs. Just a few raising edges later, the test bench has already sent in new input, a = 7 and b = -4, along with a new vld_in.

For the sake of understanding, ignore all other signals expect the vld_in and vld_out signals. Once the Testbench raises vld_in, the processes v<=a+b and w<=b*2 begin their atomic computation, and once finished, raise vld_out to 1. This output is used on the very next raising clock edge as inputs for the processes x<=v-w and y<=v+w. Once x and w are calculated, their vld_out is sent to z <= x*y to complete the final fifth computation. In a sequential processing system, line 2 cannot be processed until line 1 has terminated, and line 4 cannot be processed until line 3 has finished. However, in a concurrent system using control flow techniques, it is possible to run a command as soon as the resources are available. Line 2 simply requires knowledge of 'b' to run. It must not wait for variable 'v' to be calculated. While sequential systems cannot work concurrently by definition, sequential systems can use pipelining to improve their performance and become *pseudo*-concurrent.

This is the standard we will seek when looking to improve Von-Neumann and Harvard architecture systems.

5 Von-Neumann Architecture

With this foundational knowledge as context, let us turn to the Von-Neumann CPU architecture to examine its shortcomings and potential for pipelining. The key to understanding Von-Neumann is that such a system has a single memory unit, which stores both dynamic data variables and static instructions. This defining feature of a Von-Neumann system,

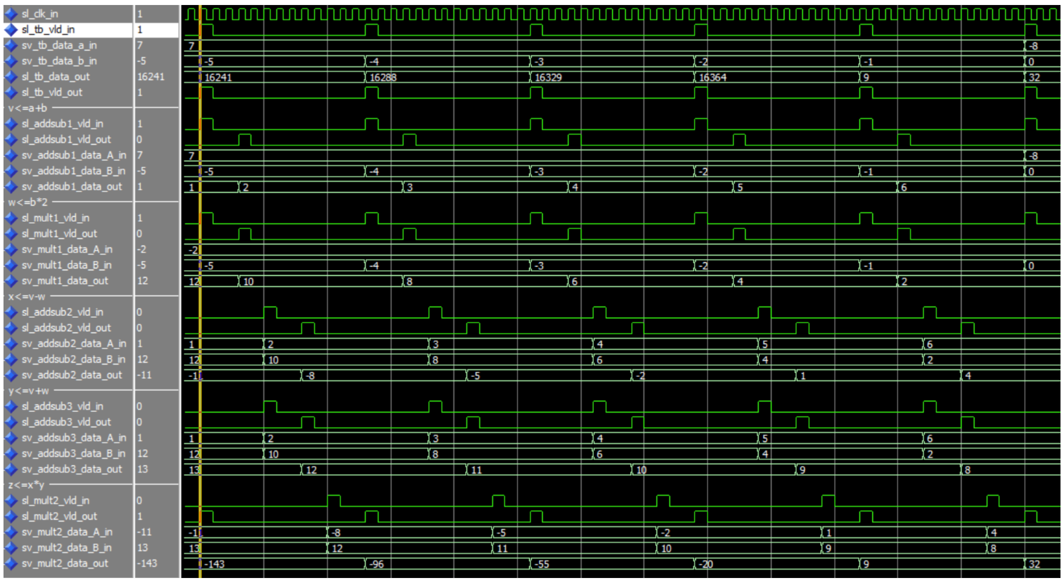


Abbildung 6: Fully pipelined VHDL implementation

however, is also a performance-inhibiting bottleneck. Keeping our wish of pipelining in mind, there are three phases in the instruction cycle that require access to the system's memory bus: (1) instruction fetch, (2) data fetch and (3) store.

Figure 7 illustrates a blocking process using Von-Neumann architecture. The OP-Code '0001', denoting addition is executed on the values stored in address '1000' and '1001' to yield the result of 25. The result is then stored at address '1000'. The process is blocking because no pipelining is taking place. In figure 8, which is a multiplication of values 2 and 15, the instruction is '0100100110011101'. The values found at addresses '1101' and '1001' are multiplied and stored at address '1001'. It takes about 11 cycles for the instruction cycle to complete.

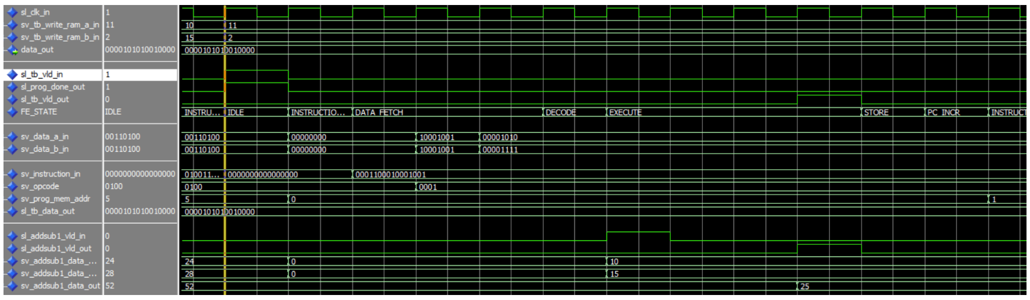


Abbildung 7: Timing diagram for command v <= a + b in Von-Neumann

Now that we have a grasp of the preliminary shortfalls of the Von-Neumann system, let us take a closer look at optimising the through-put. To begin the process of pipelining the target 5 lines of code from figure 1, we start at the first phase: Instruction Fetch: In the instruction fetch stage, the program counter points to the next location in memory

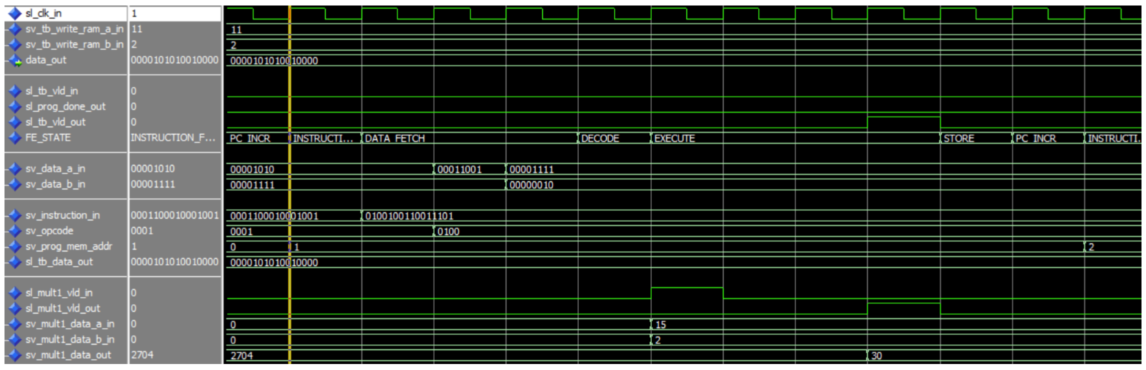


Abbildung 8: Timing diagram for command v <= a * b in Von-Neumann

where we grab the next instruction and temporarily store the instruction in a local register. Currently, the timing diagram of our Von-Neumann model in figure 7 has a single program counter at the end of the instruction cycle, where the counter value is raised after the storing order. If we wish to pipeline this phase, we need to move the program counter to the instruction fetch phase in order to drive the next input immediately after fetching. With this small design improvement, we raise the program counter to immediately again grab the next instruction thus driving input into the pipeline.

5.1 Pipelining Von-Neumann Data Fetch

Now that we have the means to drive the instruction cycle every clock cycle, let us look at second longest phase in the instruction cycle: data fetch. Figure 7 shows that the data fetch stage requires 3 clock cycles while the instruction fetch requires just one. Note as well that the output register of the instruction fetch must remain constant for the duration of the data fetch, as the data fetch stage will use the values in this register to grab data variables from memory.

With this example, we have run into the main roadblock in pipelining a system: data bottlenecking. A chain of sequential processes that have been configured to be pipelined is always bottlenecked by its slowest process. Loading instructions on every clock cycle has no use if the next process in the line needs longer to process that data. The data fetch stage entails dereferencing operands in memory to find their value. We therefore inherently have little room for optimising a system whose individual processes require varying execution times. One improvement to be made is by making the ram memory dual, triple or even quad port to enable dual, triple or quad look-ups simultaneously. Note that all examples in this paper already feature a dual ported main memory, as shown in the code snippet below.

```
sv_data_a_in <= ram_data_port1(ti_u(sv_dereference_A))(8-1 downto 0);
sv_data_b_in <= ram_data_port2(ti_u(sv_dereference_B))(8-1 downto 0);
```

Another helpful tool to pipeline instruction fetch into data fetch is to use an arbiter to control the flow of data. Since 3 instruction fetches are possible for each data fetch, we need three output registers where the instructions are temporarily stored in a round robin fashion, instead of using just a single register. We then use an arbiter to control the multiplexing of the three instruction registers for the input of the data fetch. The arbiter monitors which instruction fetch output register has been read and can therefore be overwritten with new information. If no register has been read and is thus not free, instruction fetch will read data_fetch's rdy_out as '0' and thereby wait.

5.2 Optimising ALU Execution

Having pipelined instruction fetch into data fetch, we turn to the ALU's execute phase, which requires 4 clock cycles for either subtraction, addition or multiplication. Since execute is the longest phase, pipelining and optimising all other phases is pointless if the ALU is always lagging behind in output. As mentioned earlier, a pipelined system is only as fast as its slowest component. The optimisations of the previous paragraphs mean nothing if the ALU bottleneck remains. We should therefore introduce a fifo queue between data fetch and execute, where tasks can be deposited and consumed. When the fifo becomes full, because the ALU needs more time to consume its queue of tasks, this will trigger an interrupt to avoid overwriting valid information so that the ALU can catch up and finish its tasks. If its fifo is full, it outputs rdy_out = '0', thereby signaling to the decode phase to stop and wait for the ALU to catch up. In general, the rdy_out output of each step in the instruction cycle is connected to the input of the previous component. A fifo is, however, not a perfect solution to improving ALU performance. It simply delays an impending bottleneck. Therefore, we need a better solution.

An ALU comes pre-programmed with a toolkit of hard-coded arithmetic and logical operations: for example, multiplication, division, boolean functions, etc. Given an ALU is atomic and blocking, the only option to improve ALU speed is for a CPU to have multiple ALUs for a given instruction and seperate ALUs for each instruction.

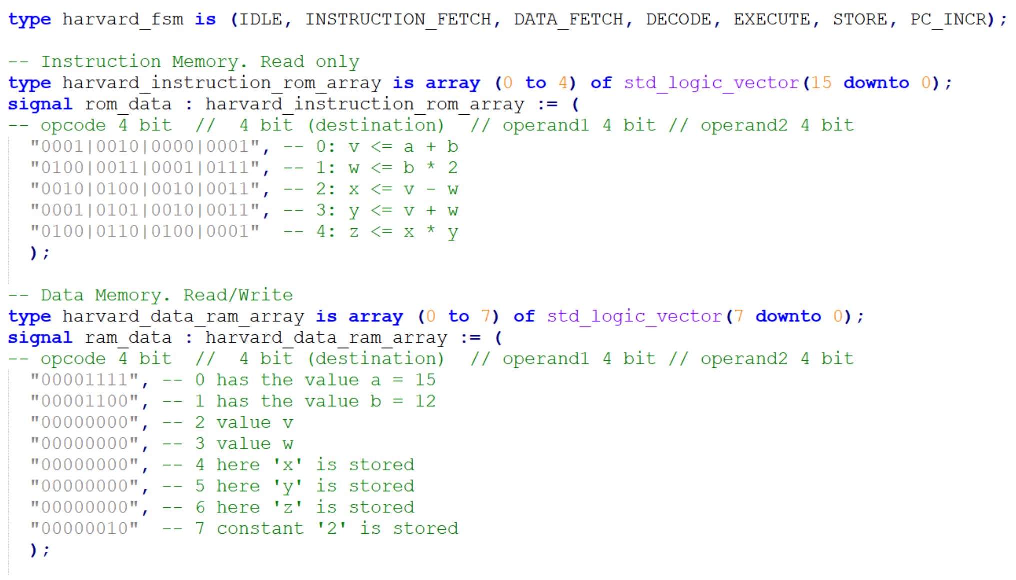
With multiple ALUs specialised in, for example, 32-bit division, 16-bit addition or single operand incrementation, we can run parallel division functions. By having several ALUs specialised in each arithmetic function with which the ALU comes hard-coded in, we improve through-put significantly. This rules applies for sequential logic. Allowing parallel computation of combination logic in the ALU is fairly straight forward to implement. As combination logic deals with logical gates, we simply need 'x' amount of identical swimlanes, equiped with an array of AND/OR/NAND, etc gates to compute with.

Note that these recommendations are not specific to Von-Neumann architecture. Because the Harvard architecture system is based off Von-Neumann systems, the optimisations in the last chapters apply to Harvard as well.

6 Harvard Architecture

Note the instruction cycle between between Von-Neumann and Harvard is identical. Only when striving for an ideal pipelined system does Harvard offer concrete advantages. The defining difference between Von-Neumann and Harvard systems on a basic level is its memory architecture: Harvard stores program instruction in a ROM memory while program data occupies a separate Read and Write memory. Figure 9 illustrates how to create a simple ROM and RAM memory container in VHDL. Upon the completion of an instruction, for example v <= a + b at rom_data(0), v is written to the address ram_data(2).

In Von-Neumann systems, storing and fetching instructions was not possible, but with



```
type harvard_fsm is (IDLE, INSTRUCTION_FETCH, DATA_FETCH, DECODE, EXECUTE, STORE, PC_INCR);

-- Instruction Memory. Read only
type harvard_instruction_rom_array is array (0 to 4) of std_logic_vector(15 downto 0);
signal rom_data : harvard_instruction_rom_array := (
-- opcode 4 bit  //  4 bit (destination)  // operand1 4 bit // operand2 4 bit
   "0001|0010|0000|0001", -- 0: v <= a + b
   "0100|0011|0001|0111", -- 1: w <= b * 2
   "0010|0100|0010|0011", -- 2: x <= v - w
   "0001|0101|0010|0011", -- 3: y <= v + w
   "0100|0110|0100|0001"  -- 4: z <= x * y
   );

-- Data Memory. Read/Write
type harvard_data_ram_array is array (0 to 7) of std_logic_vector(7 downto 0);
signal ram_data : harvard_data_ram_array := (
-- opcode 4 bit  //  4 bit (destination)  // operand1 4 bit // operand2 4 bit
   "00001111", -- 0 has the value a = 15
   "00001100", -- 1 has the value b = 12
   "00000000", -- 2 value v
   "00000000", -- 3 value w
   "00000000", -- 4 here 'x' is stored
   "00000000", -- 5 here 'y' is stored
   "00000000", -- 6 here 'z' is stored
   "00000010"  -- 7 constant '2' is stored
   );
```

Abbildung 9: VHDL view of memory architecture in Harvard System

Harvard, separating dynamic from static data allows us to simultaneous store data and retrieve instruction. In fact, all phases of the instruction cycle can occur simultaneously, except for Data Fetch and Store. These two cycles access the same container, so by nature they may not occur simultaneously (although some memory containers, i.e. Xilinx Ultrascale+ FPGA block ram, allow to specify read before write or write before read when a collision occurs Xilinx (2021)).

6.1 Improved pipelining of instruction and data fetch

In Von-Neumann systems, fetching instructions and data was only possible sequentially. Even with a separate instruction and data memory in Harvard systems, concurrent pipelined execution is not possible. However, with a small tweak, these tasks can indeed occur *p*seudo-simultaneously. If our Harvard read only instruction memory is dual ported, we can grab instruction address N and N+1 at once. In the same clock cycle, our Harvard data memory (configured to be quad ported so that it may dereference four variables, i.e., two lines of code simultaneously) is dereferencing 2 operands from instruction N-1 and 2 operands from instruction N-2. While we cannot fetch data for instructions we do not yet have, we can fetch data for instructions received in the last raising clock edge. As seen in figure 6, 7 and 8 timing diagrams, a memory look up requires a single clock cycle and is thus possible.

Implementing this strategy conquers the first major bottleneck in the instruction cycle between instruction and data fetch. Unfortunately, Harvard systems do not offer a solution to the ALU bottleneck, so there are no further improvements to be made in the execution phase.

7 Conclusion

The combination of a Hardware Description Language (HDL) and simulation software is a powerful tool to investigate the functionality of real-world hardware systems. One can either read up on the topic of pipelining and parallel processing in books or start up Modelsim and model the behavior of signals themselves in processing units. The latter method provides a hands-on opportunity to fully understanding these complex machines.

Since the 1970s, Moore's law has successfully predicted the course of IC performance. However, Moore's law's exponential curve is slowly flattening as transistors cannot physically be made any smaller. Thus, hardware designers are having troubling finding new ways to improve performance. One method to improve performance without adding more transistors to an IC is to more efficiently use existing technologies via pipelining. Through a combination of various digital design flow control tools on both Harvard and Von-Neumann CPU Architecture systems, i.e. arbiters, fifos, caches, additional memory ports and additions to component's interfaces (vld_in / vld_out and rdy_out), the potential for pipelining and thus overall increased output is great.

The number of transistors per chip cannot forever increase. With a fully concurrent FPGA system as our guide, it is clear to see that the next frontier for increased performance lies in increasing efficiency.

Literaturverzeichnis

Pipelining. (2019). Retrieved July 15, 2023, from https://www.elektronik-kompendium.de/sites/com/1705221.htm.

Xilinx. (2021). *Ultrascale architecture memory resources.* Retrieved July 15, 2023, from https://www.xilinx.com/support/documents/user_guides/ug573-ultrascale-memory-resources.pdf.

Abbildungsverzeichnis

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

[illegible]

Regensburg, den 28.07.2023

Reilly Ertman