5-2019

Investigation of the Benefits of Interlocked Synchronous Pipelines

Sabrina Rose Levitan
srl8049@rit.edu

INVESTIGATION OF THE BENEFITS OF INTERLOCKED SYNCHRONOUS PIPELINES

by
Sabrina Rose Levitan

GRADUATE PAPER

Submitted in partial fulfillment
of the requirements for the degree of
MASTER OF SCIENCE
in Electrical Engineering

Approved by:

_____

Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

_____

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
MAY 2019

To my family for supporting me throughout my time at Rochester Institute of Technology.

Abstract

The majority of today's digital circuits use synchronous pipelines. As the technology nodes get smaller, these pipelines are facing problems with area, power, and timing. One of the major sources of power consumption is the global clock and stall signals. These signals have to be routed across large sections of the chip, and with regards to stalling the pipeline, often face significant timing issues. One solution, developed by Hans M. Jacobson et al., is "Synchronous Interlocked Pipelines". This pipeline design combines synchronous pipelines with the handshaking of asynchronous pipelines. Asynchronous pipelines are less power intensive because they send acknowledge and request signals to neighboring stages that allow stages to turn off when not being used. Jacobson et al. use this handshaking technique to create local valid and stall signals instead of using global ones. To test the benefits of this design, an asynchronous pipeline, synchronous pipeline, and interlocked synchronous pipeline were built using a generic 45 nm library. Comparisons showed that while the asynchronous and interlocked synchronous pipelines took up 4 times more area than the synchronous pipeline, the asynchronous pipeline had the highest throughput of the three pipeline designs, followed by the interlocked synchronous pipeline. The synchronous pipeline had the worst throughput.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This paper is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Sabrina Rose Levitan

May 2019

Acknowledgements

I'd like to thank my advisor and professor Mark Indovina for helping me and supporting me throughout my last two years at RIT.

I'd also like to thank my parents for always checking in on me and making sure I got to work and started writing this paper.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Pipelines are an integral part to today's circuits. By breaking up data paths into smaller stages, they increase throughput and efficiency. Pipelines can be designed to operate asynchronously or synchronously. Asynchronous pipelines have latches that are clocked by handshaking signals. The handshaking signals tell each individual stage when to start computing its data. Synchronous pipelines clock every latch by using a global clock network. Depending on the type of latch used (individual or a master-slave setup), the latches are either clocked on the same clock edge or the stages alternate between the positive and negative edge.

Synchronous pipelines are used in the majority of circuits. However, as transistors have gotten smaller, synchronous pipelines have been becoming more problematic. Global clock networks have to stretch across the entire chips. Wire delays are increasing, so these clock networks are slowing down the circuit and using up lots of power. In addition, signals used to stall sections of the pipeline take too long to reach the latches they need clock, which results in lost data.

To combat these problems, some designers have returned to looking at asynchronous pipelines. Since latches in these pipelines are clocked only by the latches directly adjacent to them, long

stall signals don't exist. There is also no global clock network to eat up power. Overall, asynchronous pipelines consume less power than synchronous ones. Latches are only active when there is data to compute, and therefore don't waste power storing unnecessary data.

Since current technology is based around synchronous pipelines, switching to asynchronous pipelines would be a long and difficult process. The biggest issue is that most of the tools used for developing large digital circuits only produce synchronous pipelines, and would have to be reworked to make the switch. This would take a lot of time and money.

Jacobson et al. [11] solved this problem by designing the interlocked synchronous pipeline. This pipeline adds asynchronous elements to synchronous pipelines, thereby utilizing the existing development tools while reaping the benefits of asynchronous pipelines. The aim of this paper is to explore the interlocked synchronous pipeline by comparing it directly to both asynchronous and synchronous versions of the same pipeline.

The structure of the paper is as follows: Chapter 2 is a literary review of pipeline research. Chapter 3 gives an in depth operation of each type of pipeline. Chapter 4 explains the pipeline design chosen for the comparison and the gates developed for it. Chapter 5 shows the layouts for each pipeline and explains the choices behind the layout structure. Chapter 6 goes over the results from testing each pipeline. Chapter 7 discusses what these results mean in a larger context. And finally, Chapter 8 concludes the paper.

Chapter 2

Literary Review

Asynchronous pipelines were first conceived of by David Muller in 1963 [12]. They can be broken down into two categories: static and dynamic. Static pipelines use latches to store data and handshaking logic to pass the data between the latches. The handshaking logic must be designed to meet the timing of the data path between each latch. Dynamic pipelines use a four-phase handshaking protocol to pass data between logic gates, skipping latches entirely. The lack of latches causes the dynamic pipelines to be faster, but they require more design work and are more susceptible to noise [12].

The general structure of a static asynchronous pipeline is shown in Figure 2.1.

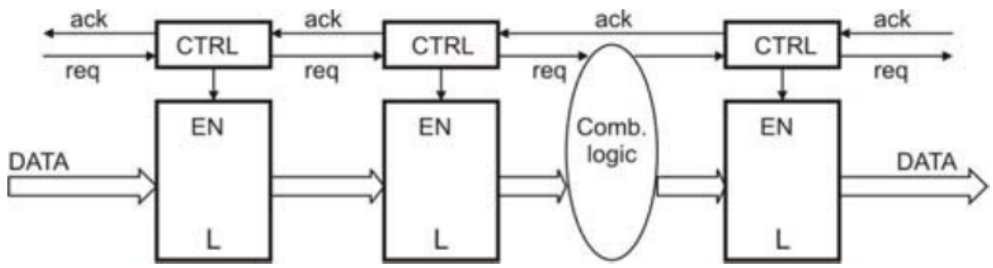

Figure 2.1: Block diagram of an asynchronous pipeline [1]

Each stage of the pipeline is made up of a latch, control logic, and optionally combinational logic. The control logic is used to process the two handshaking signals: request and acknowledge. There are several different handshaking protocols.

The first protocol is single rail encoding. The "single rail" name is due to the fact that the handshaking signals are only one bit, either low or high. One wire is used for the data and another is used for the request signal. The request signal acts as a strobe to signal to the receiving latch that data is coming. The timing between the data being sent and the request signal going high must be carefully matched. When the receiver receives the data, it sends back an acknowledge on the same wire that the request signal used [1, 12].

Single rail encoding is used with the two phase handshaking protocol, which operates by using the rising edge of the request signal and the falling edge of the acknowledge signal to communicate that data is ready to be sent.

The other common protocol is four phase handshaking. As the name implies, this protocol has four phases:

1) The request signal is set high to tell the receiving latch that there is data to transmit. The latch sends the data to the receiving latch.

2) The receiving latch accepts the data and sets the acknowledge signal high.

3) The request signal is set low.

4) The acknowledge signal is set low.

The four phase protocol is simpler to implement than the two phase protocol, but requires extra power to set the request and acknowledge signals low after every data transmission [1, 2, 12].

This "return to zero" procedure can be hidden by implementing a Dual Control Path, which uses two control paths for the control logic. While one path is in the "return to zero" phase, the other one can continue to drive the request and acknowledge signals. Then, when that one

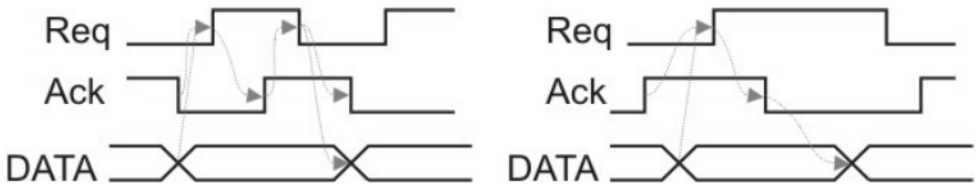

Figure 2.2: 4 phase protocol (left) and 2 phase protocol (right) [1]

moves to the "return to zero" phase, the first control path takes over [13]. While this method increases the throughput and timing of the pipeline, it does add additional area and control logic.

Figure 2.2 shows the waveforms for the 4 phase protocol and the 2 phase protocol.

To coordinate the handshaking signals, gates called Muller C-Elements are used. C-Elements use the request and acknowledge signals to generate clock pulses for the latches. If the two inputs to the C-Element are the same value, the output will be that value. If the inputs are different, the C-Element retains its last value.

Figure 2.3 shows a CMOS implementation of the C-Element. The inputs are A and B and the outputs are C and C bar. Transistors M1 and M2 act as an AND gate and transistors M3 and M4 act as an OR gate. M5 and M8 act as a latch to store the data [2].

While the Muller C-Element is common for asynchronous pipelines, other alternatives exist. The GasP pipeline, which stands for Go Asynchronous Symmetric Pulse Protocol [14], uses the circuitry shown in Figure 2.4. GasP is a high speed design but requires complicated timing work to ensure that the data and control paths work properly together [15].

The circuitry consists of the handshaking control and the latch itself. In this figure, L is a signal generated by circuitry to the left, R is generated by circuitry to the right, A is an internal signal, and LE is the latch enable. The latch is the transistor and two inverters at the bottom of the figure [3].

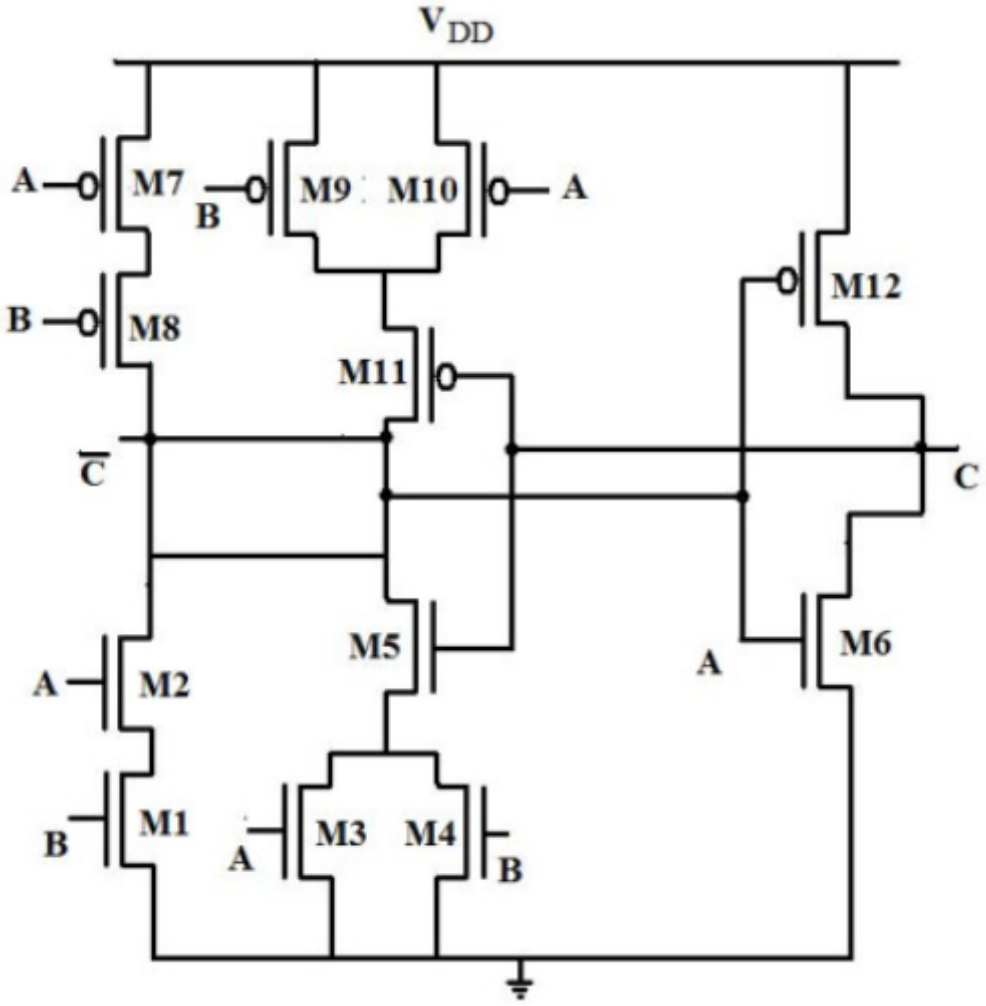

Figure 2.3: CMOS C-Element [2]

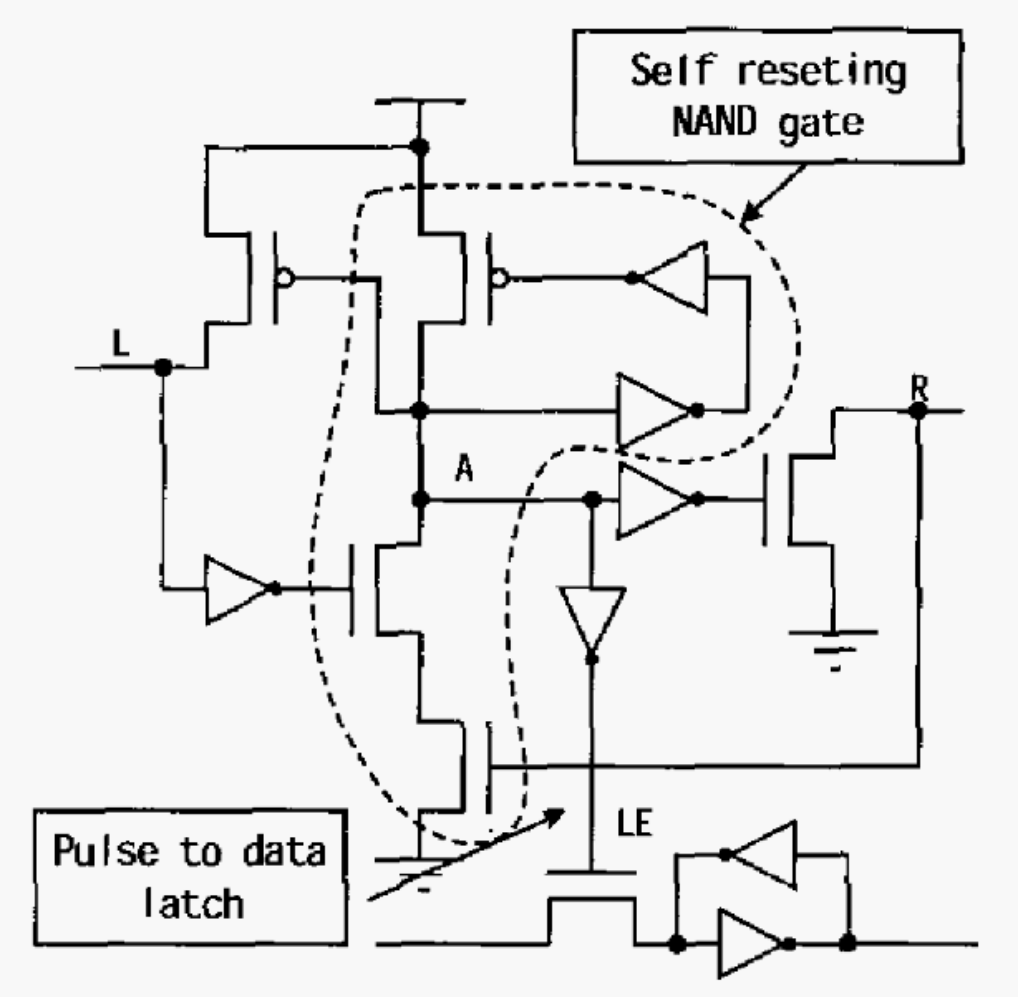

Figure 2.4: GasP Circuitry [3]

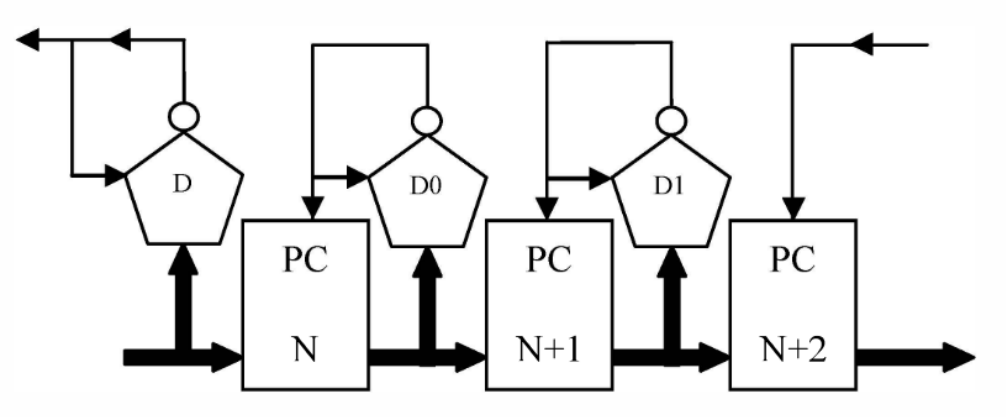

Figure 2.5: Self-Controlled Pipeline [4]

The main component is the self-resetting NAND. When L goes low, the NAND will trigger the latch to store the incoming data. When R goes low, the NAND will disable the latch. It will not re-enable the latch until R goes high to signal that the circuitry to the right is done with the data [3].

Dynamic pipelines use their own handshaking logic circuits called completion detectors (CD). Figure 2.5 shows an example of a dynamic pipeline called the Self-Controlled pipeline.

The pipeline is made up of logic blocks, the rectangles, and CDs, the pentagons. The logic blocks take in a precharge signal from the CD. The CD uses an asymmetric C-Element. This C-Element is similar to the Muller C-Element but it has three inputs. The inputs are called "+", "-", or unmarked. If the unmarked and "+" signals are high, the output is high. If the unmarked and "-" signals go low, the output is low. For other combinations, the output stays the same, as it did with the Muller C-Element. The output of the C-Element is the "done" signal, which states if the logic block has finished computing. If it has, it can accept new data [4].

A similar pipeline, the Self-Precharging pipeline, is presented in [16]. The done signal from the CD is used to precharge the dynamic logic in the logic block as well as the CD itself [16].

Another dynamic asynchronous pipeline is the high capacity pipeline. The pipeline design involves modifying dynamic logic gates so that they are able to store data. This is done by decoupling the pull-up and pull-down transistors. The pull-down transistors are controlled by the evaluate signal and the pull-up transistors are controlled by the precharge signal. Normal dynamic logic has both transistors controlled by the precharge signal [17].

If neither the evaluate nor precharge signals are high, the dynamic logic is in a new state called "isolate". When in the isolate state, the logic gate cannot accept any new inputs. By putting gates into this state, the pipeline is able to store data without using latches [17].

The High Throughput High Capacity pipeline proposed in [5] builds upon this idea using the communication protocol shown in Figure 2.6.

Each stage goes through the evaluation phase, the isolate phase, and the precharge phase. When stage N finishes its evaluation phase, the CD puts it into the isolate phase. When stage N+1 finishes its evaluation stage, its CD puts it into the isolate phase and also sends a precharge signal back to stage N [5]. This precharge signal is effectively the acknowledge signal, saying that stage N+1 has received the data that stage N is currently storing, so stage N can move on and accept new data.

Asynchronous pipelines can also be used in wave pipeline designs. Wave pipelines are pipelines don't use latches. Instead, the logic gates within the wave pipelined section are designed so that logic can propagate through them in a "wave". The timing of the gates is carefully chosen so that data can move through it without getting overwritten by another wave of data behind it.

For an asynchronous wave pipeline, the request and data signals are physically linked together to keep them in sync. The request signal is used to control dynamic latches built out of transmission gate muxes, which use a similar principle of storing data in dynamic gates as the previous examples. These latches are inserted between every logic gate. The designers of the

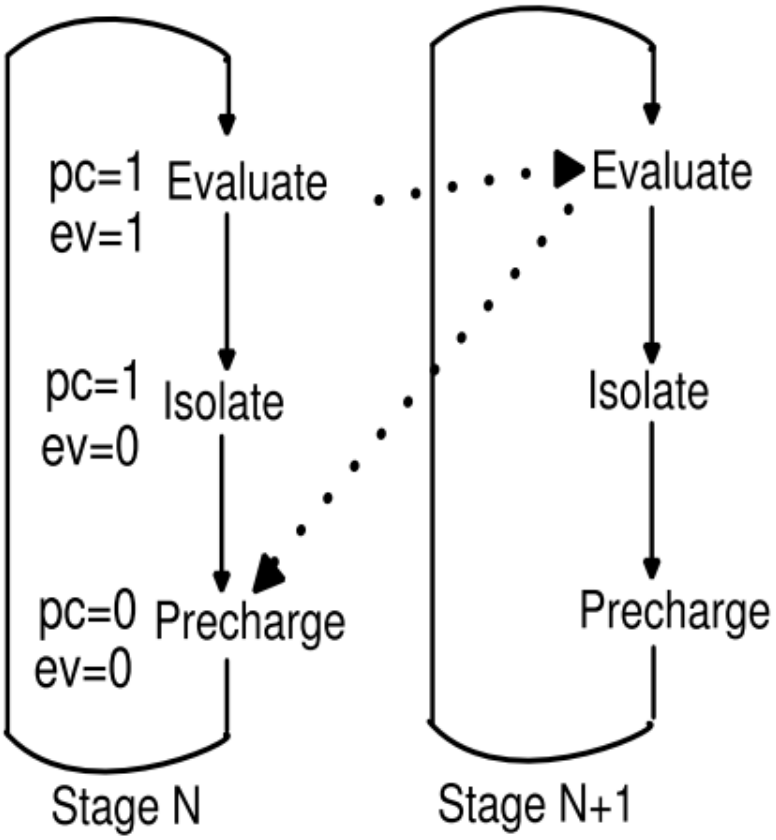

Figure 2.6: High Throughput High Capacity communication protocol [5]

pipeline argue that while latches are normally not allowed in wave pipeline, a pipeline with only one gate between latches is inherently a wave pipeline because “the fine granularity keeps the waves coherent” [18].

On a bigger scale, asynchronous pipelines can be used to build microprocessors. While most of today’s processors are synchronous, several different types of asynchronous processors have been developed. One such processor is the Counterflow Pipeline Process (CFPP), developed in 1994. The main feature of this processor is that information flows in two directions [6, 19, 20]. This can be seen in Figure 2.7.

Instructions start by the program counter at the bottom of the figure. Each instruction consists of the opcode and the source and destination register information. The processor reads the source information and the values from those registers are sent down the pipeline from the register file. When the instruction and operands meet at a stage, the operation takes place. The instruction, which now contains the resulting value, continues towards the register file and eventually writes its value into the destination register. It also places the value into the results pipeline so it flows down towards the program counter. This way, if a new instruction needs that value, it will encounter it earlier in the pipeline than if it had to be fetched from the register file [6].

Alongside the two pipelines are functional units used to perform the instruction operations. These functional units, called “sidings”, are also pipelined. The first stage is a “launch” stage, used to move the instruction from the instruction pipeline into the siding. The final stage is the “return” stage, which returns the instruction to the pipeline. In between are stages used to do the computation. While sidings are not required for the CFPP, as operations can be performed by the main pipeline stages, they allow other instructions to be processed at the same time [19].

Another asynchronous processor is the A8051. This was developed to match the Intel 8051 synchronous processor. The A8051 is a five stage complex instruction set computer (CISC)

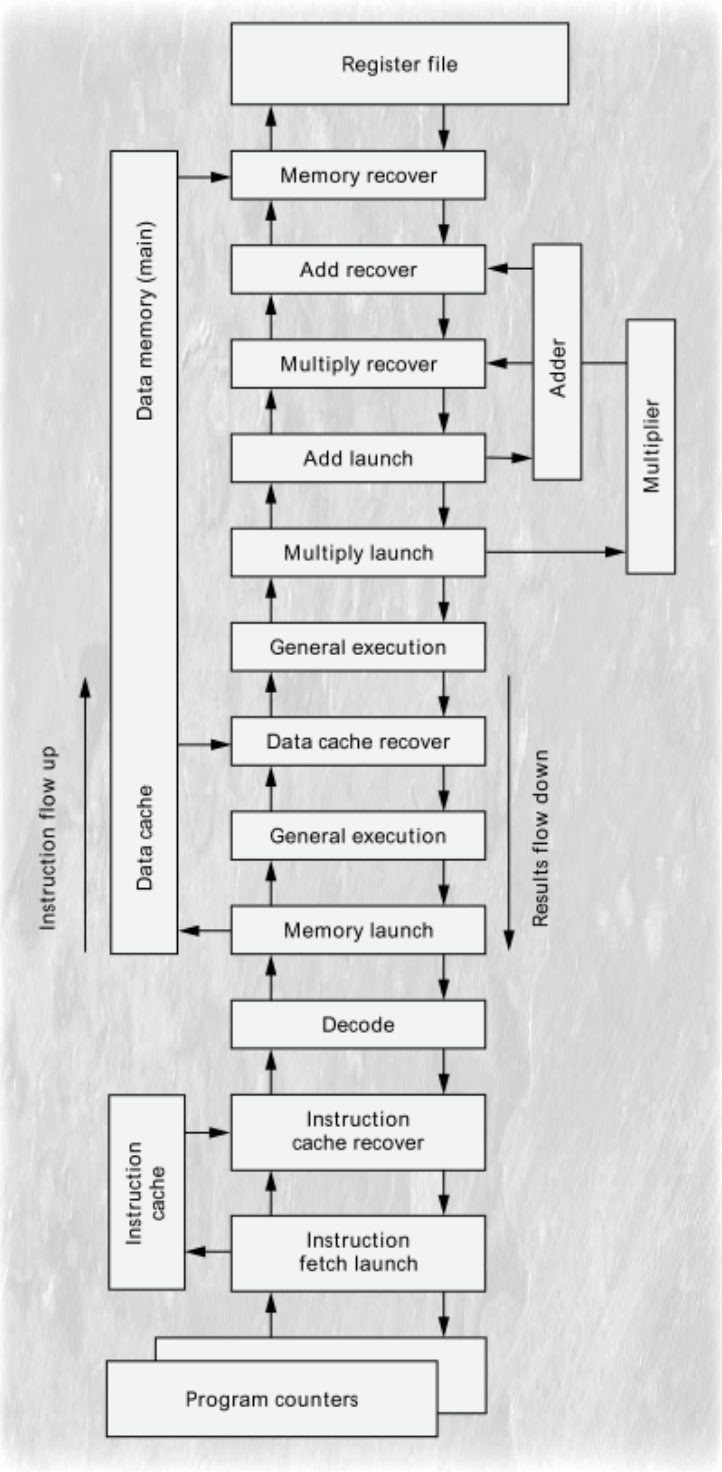

Figure 2.7: Counterflow Pipeline Processor block diagram [6]

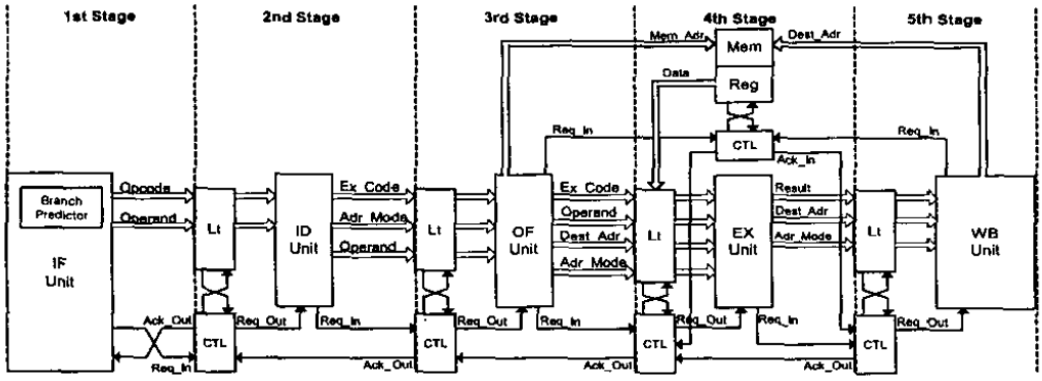

Figure 2.8: Architecture of the A8051 [7]

processor. The five stages are instruction fetch, instruction decode, operation fetch, execution, and writeback. The processor has the ability to loop the operation fetch and execution stages for longer instructions. In comparison, the Intel processor has a longer pipeline to accommodate the longer instructions, which means there are often times when stages are sitting idle [7]. Figure 2.8 shows the architecture of the A8051.

A different approach to asynchronous processors is Globally Asynchronous, Locally Synchronous (GALS). GALS circuits consist of synchronous blocks that communicate asynchronously with each other. A GALS module is shown in Figure 2.9. The synchronous module is "wrapped" in a controller, which generates a clock based on the request signal of the previous block [4, 21].

One issue with merging asynchronous and synchronous blocks like this is "synchronization failure". This is when a clock edge from the synchronous module occurs at the same time that data arrives from the asynchronous module. This may cause the circuit to enter a metastable state for an unknown period of time. Since a metastable state is a value between 0 and 1, the logic downstream could interpret it differently. This can lead to unwanted states occurring or incorrect data [22].

One solution to the synchronization problem is to use an asynchronously triggered ring

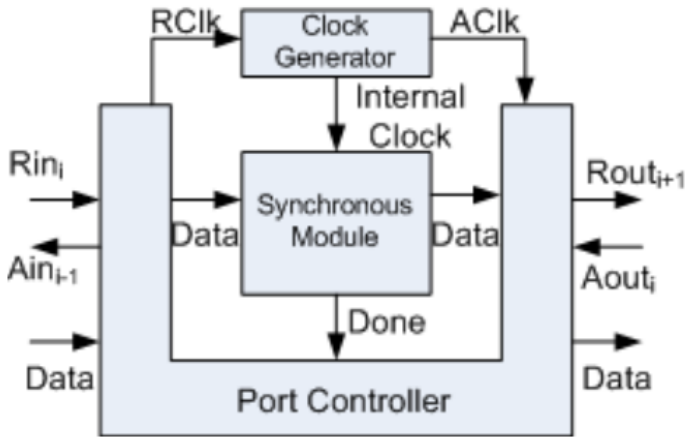

Figure 2.9: A GALS module [8]

oscillator to control the clock. The asynchronous module uses a RUN signal to turn the synchronous clock on and off. This allows the asynchronous module to disable the synchronous module when a metastable event occurs. Once the event is over and the data reaches a known state, the clock can be turned back on [22].

Modifying the clock is another way to merge asynchronous and synchronous pipelines. One of the major benefits of asynchronous pipelines is that the delay is based off the average delay of the data path, while synchronous pipeline delay is based off the worst path delay [22]. Asynchronous pipelines are more efficient because they compute the clock pulses for their latches locally so that latches are enabled as soon as data is available.

The VariPipe [23] creates a clock with an adjustable frequency. The clock period changes every cycle to accommodate the current worst case delay. This allows data to move quickly through faster data pipes instead of being forced to wait because the clock is set slower for a currently idle portion of the circuit. When that idle part becomes activate, the clock will return to the slower frequency so the data path has time to complete its work [23].

Another option for dealing with the synchronous clock problem is to use double edge

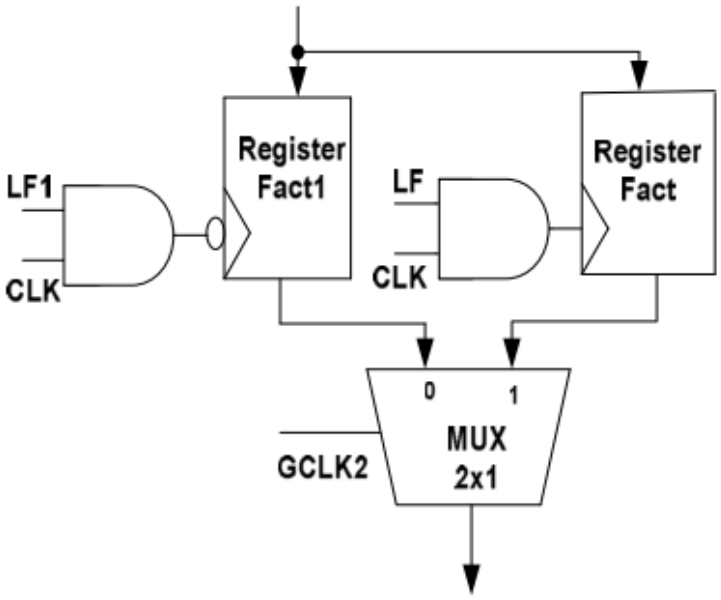

Figure 2.10: Double Edge Triggered Flop made with regular flops [9]

triggered flip flops. The flip flops, which are pairs of latches, can store data at both the positive and negative edge of the clock. This allows twice the throughput at the same clock frequency, or allows the clock frequency to be scaled down 50% without loss of performance [9].

While this idea works, it is not practical because standard cell libraries and FPGAs do not support double edge triggered flops. A workaround is to use two regular latches, one clocked off the positive edge and the other off the negative edge of the clock, and then a multiplexer to select which data to use [9]. Figure 2.10 shows this arrangement.

Instead of dealing with the clock or interfaces between asynchronous and synchronous modules, some designers have attempted to convert synchronous circuits into asynchronous ones. This would allow the designers to use the common synchronous pipeline development tools while still gaining the benefits of asynchronous pipelines.

One option is to develop the synchronous pipeline and then completely convert it over to an asynchronous design. This is done by replacing the synchronous registers with asynchronous

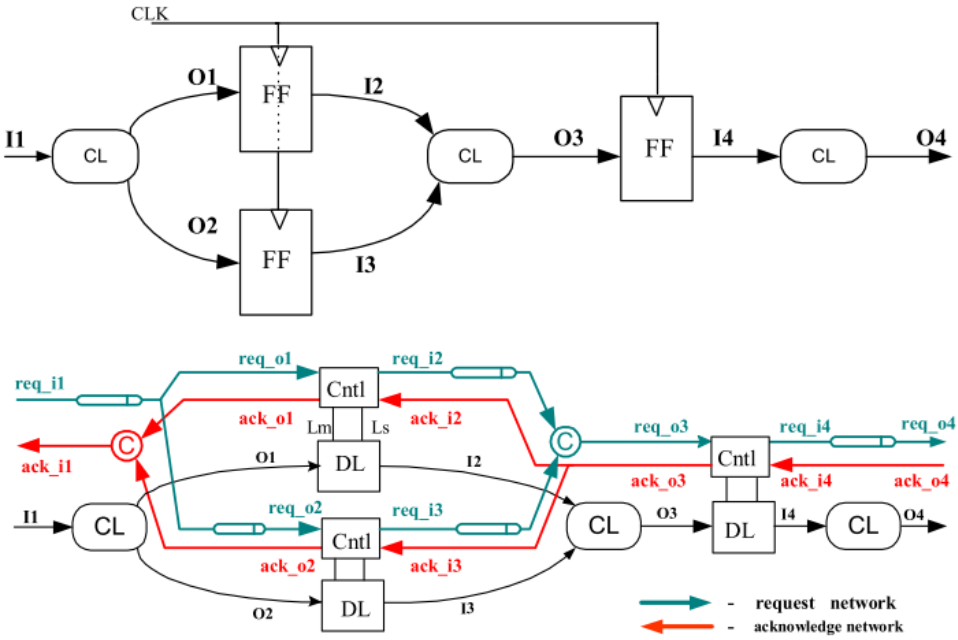

Figure 2.11: Conversion from a synchronous pipeline (top) to an asynchronous pipeline (bottom) [10]

ones and adding in the handshaking signals, while leaving the combination logic untouched [10]. The full replacement strategy can be read about in [10]. Presented here is an abbreviated version.

The conversion process starts by replacing every flip flop with a latch and a handshake controller. Then it creates a request network by connecting the request signals from the controllers together. Delay elements are added to the paths as necessary. Additional gates are inserted to deal with two paths that join together [10].

The second step is to create the acknowledge network. This is done by connecting the acknowledge signals in the reverse direction, again adding logic to handle paths that join together. This completes the conversion process [10]. An example can be seen in Figure 2.11.

A second option is to convert a synchronous pipeline into an elastic synchronous pipeline. An elastic pipeline adds handshaking signals to a synchronous pipeline. Instead of a request

signal, the elastic pipeline has a valid signal. This signal states if the data is valid or not. In the other direction is a stop (or stall) signal. This works as the opposite to an acknowledge signal and tells the previous stage to not send data [24].

One application of the elastic pipeline is in a self-timed bit serial control unit. Control units are often implemented as finite state machines. Instead, the unit can be built with handshaking signals. The control unit in this example acts as a shift register. Data is moved through the register in accordance with the handshaking valid/stall signals [25].

The elastic pipeline, also called an interlocked pipeline, will be explored in depth in the rest of this paper.

Chapter 3

Pipelines

Digital circuits are made up of data paths, where the input data travels through several logic gates and then arrives at the output. A simple example is shown in Figure 3.1:

This type of circuit is functional but inefficient. After the data has passed through the first gate, that gate sits idle until the next round of data can be entered. This is fixed by turning the data path into a pipeline by adding registers. A register is a latch or combination of latches that stores the data at the input whenever the latch is enabled. Once the input data has been "clocked" into the first latch and saved, new data can be entered into the pipeline and use the gates in the first stage while the original set of data uses the gates in the second stage. This

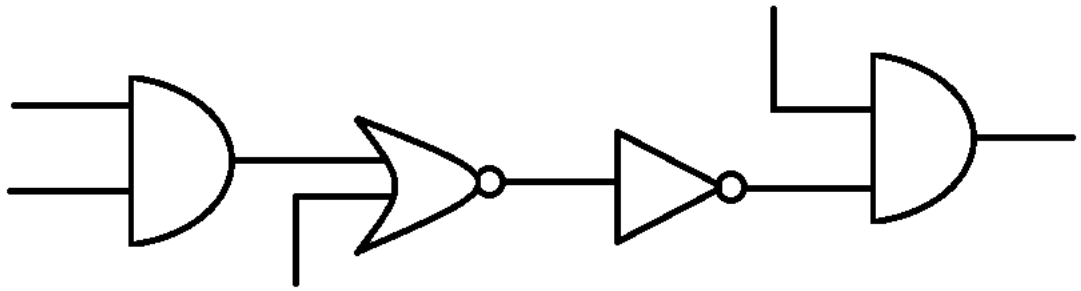

Figure 3.1: Simple Data Path

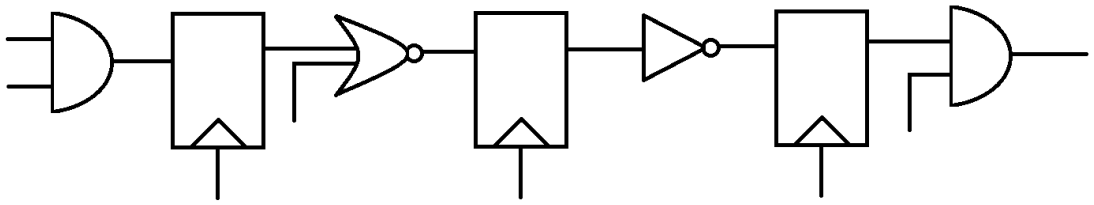

Figure 3.2: Pipelined Data Path

increases the throughput of the data path.

There are two main types of pipelines: synchronous and asynchronous. The type of the pipeline indicates how the latches are clocked. In a synchronous pipeline, all latches are clocked off the same signal and are thus synchronized with each other. In an asynchronous pipeline, the latches are clocked off local signals and therefore can be enabled and disabled out of sync with each other.

3.1 Asynchronous Pipelines

Asynchronous pipelines work by using handshaking signals. These signals connect each latch to its neighbors. The first signal is the request signal. This signal is sent by the preceding latch to say that it has data to send. The second signal is the acknowledge signal. This signal is sent by the next latch in the pipeline and says that it is ready to accept new data. Only when the request signal and acknowledge signal are both high - meaning that there is data to process and the next stage is able to accept that data - does the current latch turn on. This strategy saves power by only enabling latches when they are needed.

The handshaking signals are orchestrated by using a Muller C-element. The C-element has the following truth table:

When the request and acknowledge signals are both low, the latch turns off. When they're

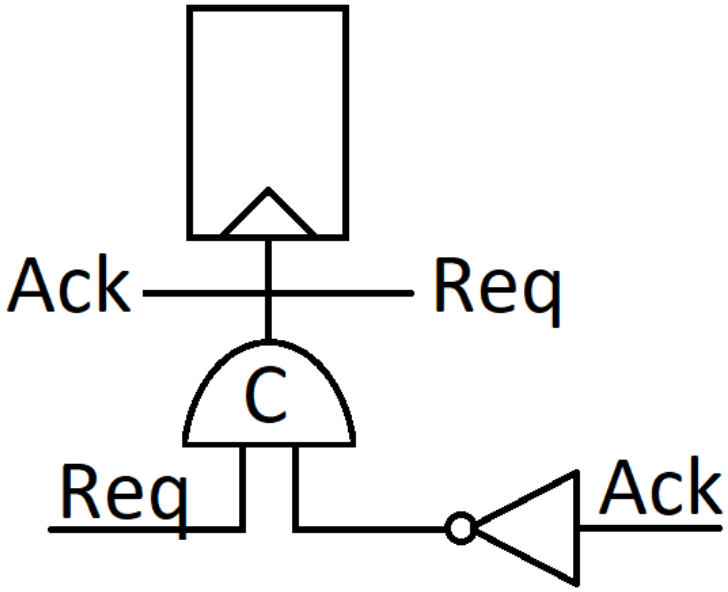

Figure 3.3: Latch with C-Element

X	Y	Z
0	0	0
0	1	Z-1
1	0	Z-1
1	1	1

Table 3.1: C-Element Truth Table

both high, the latch is enabled. When only one signal is high, the C-element remembers its last state. This means that once the latch turns on, it will continue to process data until both of its neighbors tell it to stop.

3.2 Synchronous Pipelines

Synchronous pipelines use a global clock signal to enable and disable their latches. When using a master-slave latch setup, every stage is clocked on the same edge of the clock signal. When using a single latch for each stage, the stages alternate between using the positive and

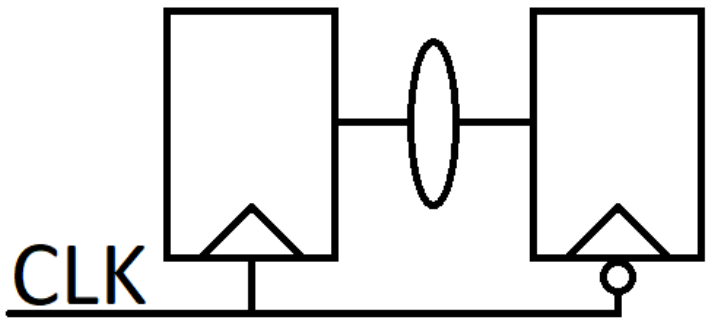

Figure 3.4: Alternate Clocking in a Synchronous Pipeline

negative edges. This is done so that when the first latch is enabled and allows data to pass through, the data gets stopped at the second latch and doesn't continue through the entire pipeline.

Since every stage is clocked off the same signal, the pipeline is forced to run at the frequency of that clock signal. The logic in each stage must be fast enough so that data can complete the path before the next clock edge. Otherwise the computation won't finish and risks getting overwritten by the new data that was just let into that stage. These timing requirements are some of the biggest problems to solve in synchronous design.

Another major issue is power consumption. In asynchronous pipelines, latches are only active when they are needed for computation. In synchronous pipelines, latches are active at every clock edge, regardless of if they are needed or not. This leads to a lot of wasted power. A common solution is clock gating. Clock gating shuts down part of a circuit by disabling the latches when they aren't needed. The signals used for clock gating often require a lot of computation, however, and have become some of the most timing critical signals in digital circuits.

Additionally, the clock gating signals face propagation issues. As they travel back through the pipeline to stall it, the signals must reach all the latches before the next clock edge arrives.

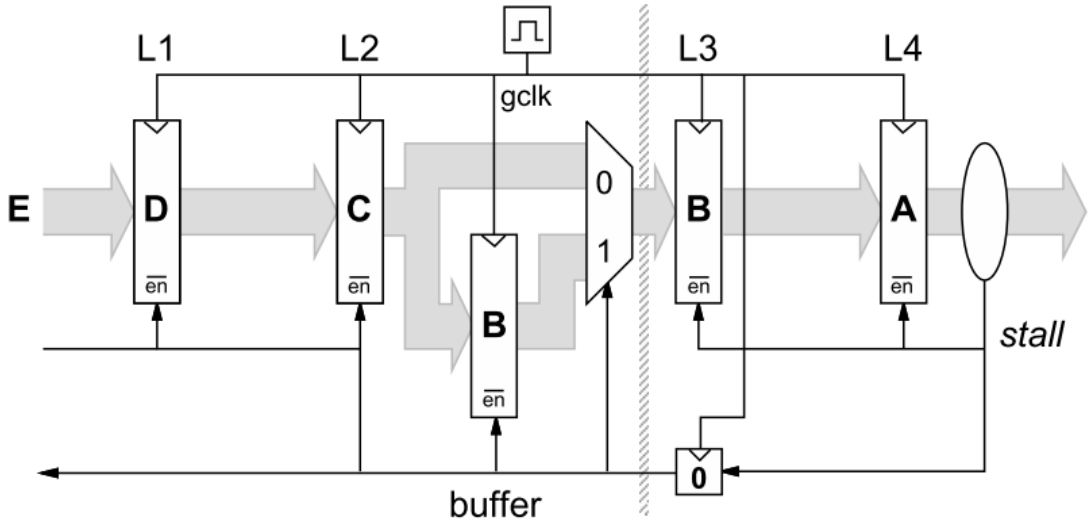

Figure 3.5: Stall Buffer within a Synchronous Pipeline [11]

Otherwise, data will get lost as one latch is stalled and keeps its data, but the preceding latch clocks in new data. This creates a "stall boundary". To fix this, stall buffers have to be added at the stall boundary. The stall buffer stores a copy of the data that might be overwritten, and a mux selects this data if it recognizes that the stall boundary caused data to be overwritten. While this solution works, it requires adding additional circuitry which increases area and power [11]. The stall buffer is shown in Figure 3.5.

3.3 Interlocked Synchronous Pipelines

The Interlocked Synchronous Pipeline (ISP) combines asynchronous and synchronous pipelines. It starts with a synchronous pipeline, where each stage is clocked on opposite edges of the clock signal. Then it adds handshaking signals that can locally stall a stage [11].

The first handshaking signal is a valid signal. The valid signal, similar to the request signal in an asynchronous pipeline, travels in the same direction as the data. It is a single bit that tells if the data is valid or invalid. This bit must be synchronized with its data, so it is also latched

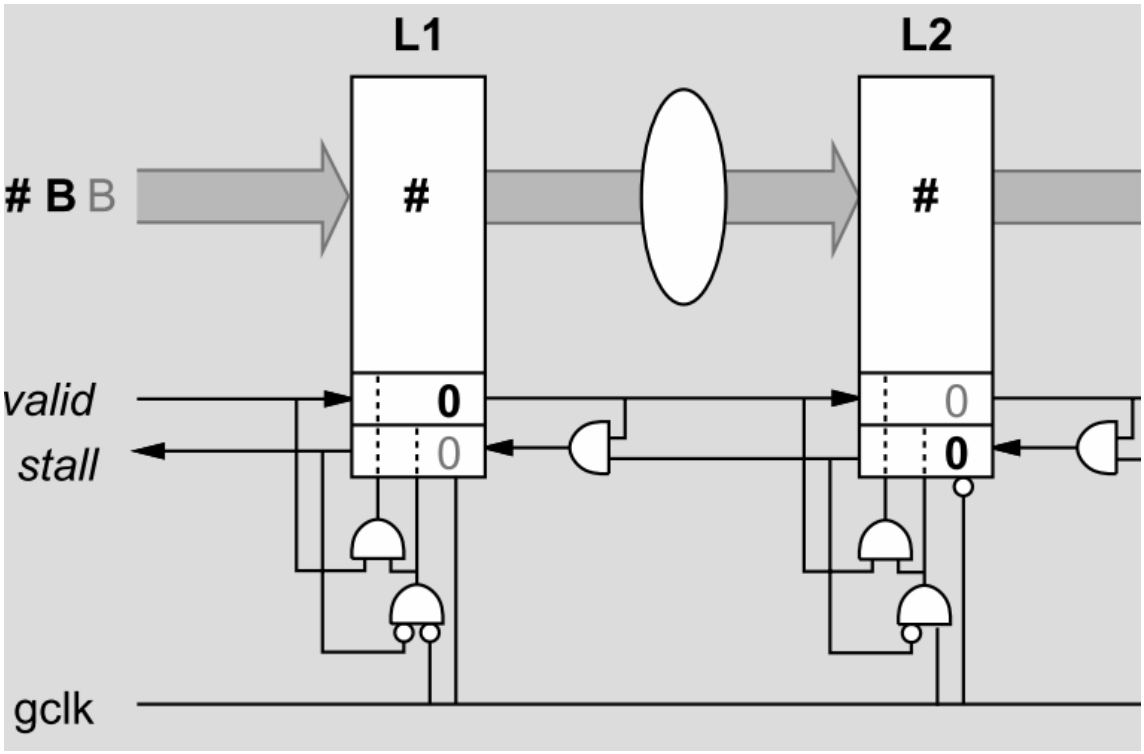

Figure 3.6: ISP Latch Stages for both edges of the clock[11]

with the data latch [11].

The second handshaking signal is the stall signal. The stall signal is similar to the request signal in the asynchronous pipeline in that it tells an individual stage if it can send data or if it has to wait. Unlike the asynchronous pipeline, however, this stall signal stays synchronized with its data and is also latched with the data and valid latches [11].

Figure 3.6 shows the clock circuitry for the three latches. The global clock signal is used to gate the stall latch. It is also combined with the output of the stall latch, and the new signal is used to clock the valid latch. This stalls the valid latch if a stall has occurred. Finally, the clk/stall signal is ANDed with the incoming valid signal to clock the data latch. The data latch only stores new data if there is no stall and the data is valid [11].

The final piece of the circuitry is at the input to the stall latch. The stall signal is ANDed

with the valid signal, clearing the stall if the data is invalid. This is one of the advantages of the ISP: if a latch contains invalid data, there is no need to stall it. Instead, valid data can continue being fed through the pipeline until it hits the stall, thereby increasing efficiency and throughput [11].

The operation of the ISP is shown in Figure 3.7. The pipeline is four stages long and consists of latches L1, L2, L3, and L4. Each latch stage shows the data it's storing (A-E) or invalid data (#). The value below the data is the state of the valid bit, and the value below that is the stall bit. The light gray areas show the invalid data. The dark gray shows the stall [11].

Data moves through the pipeline with every clock edge. When the stall starts, it travels back through the pipeline one stage at a time. When it encounters invalid data, the stall signal is cleared and the latch containing the invalid data is allowed to continue operating. This removed the invalid data in the pipeline and allows the valid B to catch up to the valid A. When B reaches the stall, its latch gets stalled and the stall continues down the pipeline. The next bit of invalid data is removed in the same way, allowing C to move forward in the pipeline. At this point the stall condition has been cleared through other means and the pipeline can continue moving. All of the invalid data has been removed and data A through C reaches the end of the pipeline. If the valid bits had not cleared the stall, the final output would have been A, #, and B. The throughput has therefore been increased [11].

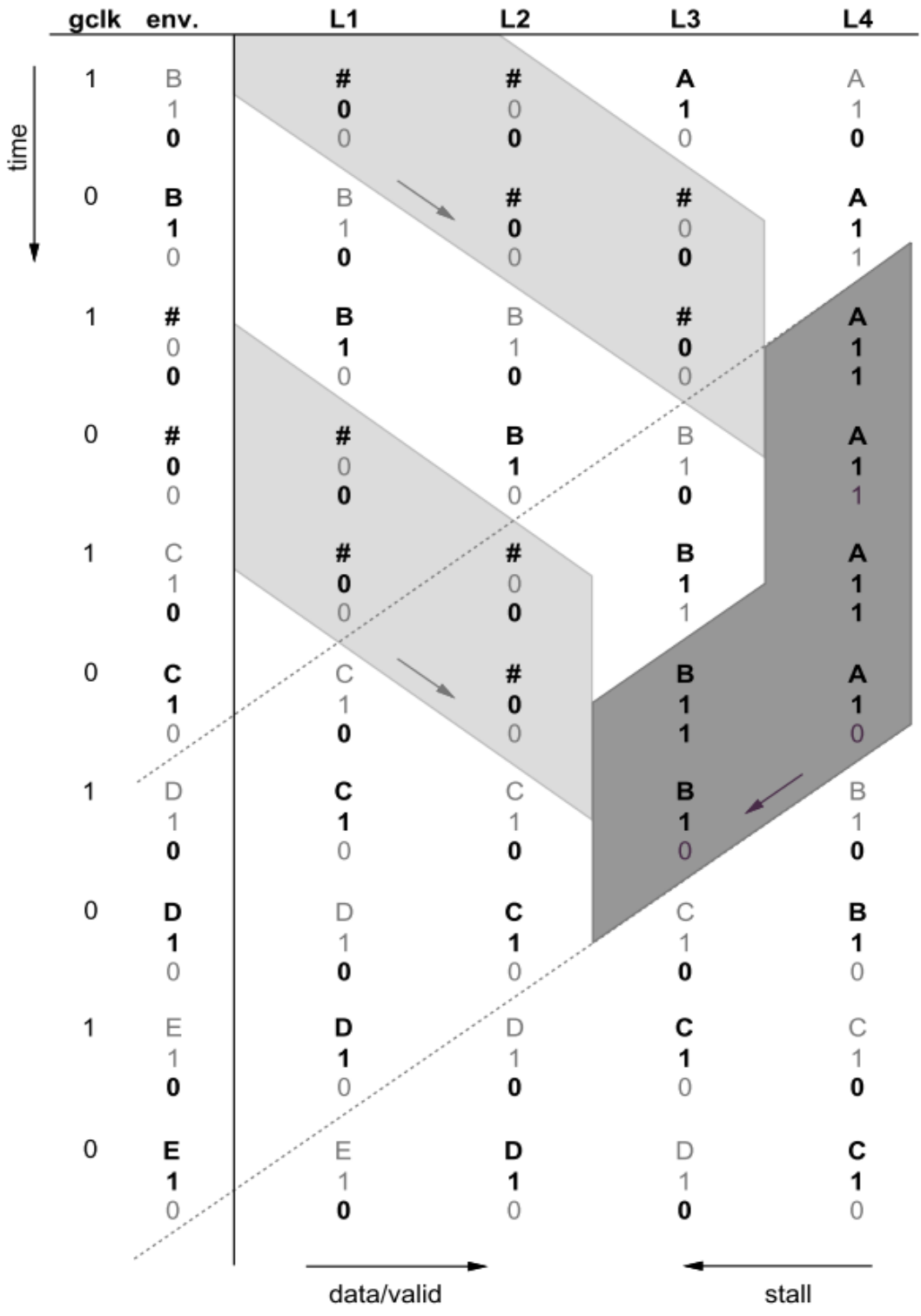

Figure 3.7: Timeline of an ISP [11]

Chapter 4

Pipeline Design

Figure 4.1 shows a simple 4 bit 4 stage pipeline. This pipeline design was created and simulated as an asynchronous pipeline, a synchronous pipeline, and an interlocked synchronous pipeline. The three pipelines were made with a generic 45 nm library. By using the same pipeline design for all three pipeline variations, their advantages and disadvantages could be directly compared.

The pipeline was designed so that each stage has a logical depth of one logic gate. This keeps timing simple, as there is no need to worry about one path being slower than another. The stall signal is generated by the logic gate for the least significant bit of stage 3. The logic gates were chosen so that a stall occurs around 30% of the time. With this probability, random inputs are likely to pass through the pipeline unhindered, but enough stalls will occur so each pipeline's stalling method can be observed. To clear a stall, a STALL_CLR signal must be asserted by the test bench.

Table 4.1 shows the truth table for the pipeline. Since the pipeline was designed to meet the 30% stall percentage, the final output values were not considered. This led to a small number of outputs: for the 16 possible inputs, there are only five possible outputs. Of these outputs,

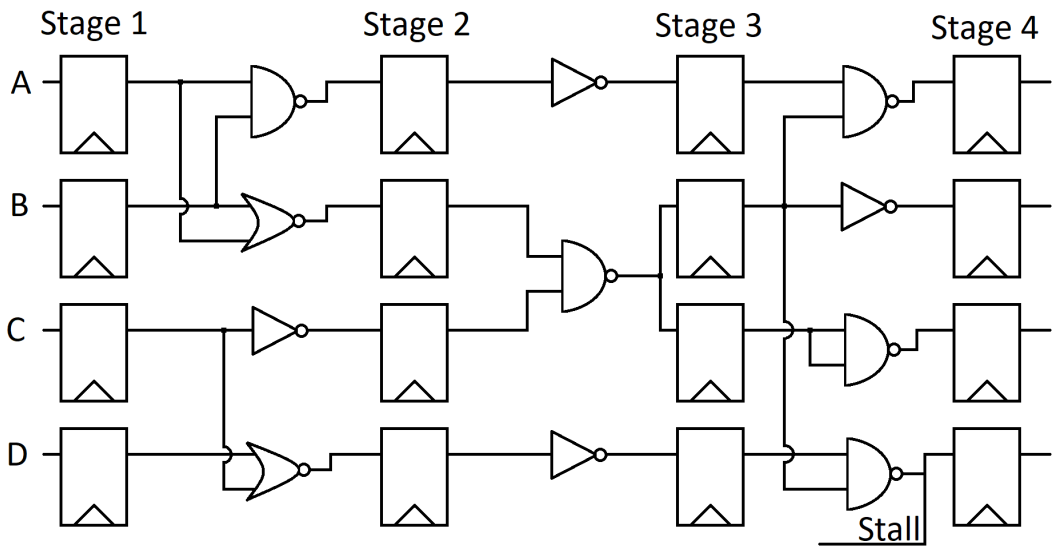

Figure 4.1: Pipeline design

bits 2 and 1 only go high for one output. This made the initial testing of the pipelines difficult. Bits 3 and 0 could be easily checked to verify that their logic and latches were correct by choosing inputs that toggled the output. Bits 2 and 1 could be checked the same way, but the only combination that set them high also set the stall. The stall had to be constantly cleared so it didn't hinder the toggling.

The other issue with the pipeline design is that the logic serves no purpose. For a given input, there is no easy way to predict the output, like there would be if the pipeline was an adder or shifted bits. This problem only affected human readability of the waveforms. Instead of knowing quickly what the expected output was, or if the observed output was correct, the input/output table had to be used.

Input [hex]	Output [hex]	Stall
0	F	1
1	F	1
2	8	0
3	8	0
4	9	1
5	8	0
6	8	0
7	8	0
8	9	1
9	8	0
A	8	0
B	8	0
C	1	1
D	0	0
E	0	0
F	0	0

Table 4.1: Pipeline truth table

4.1 Gates

Given this pipeline design, the following logic gates had to be created: INV; NAND; NOR; AND; OR; C-element; latch and latch_n. Two versions of the latch were created: one that was triggered by the positive edge of the clock (latch) and one that was triggered by the negative edge of the clock (latch_n). Designing both latches saves area; instead of using inverters to negate the clock, the latch itself is built to use the negative edge.

The latches are built using pass transistors, as shown in Figure 4.2. The pass transistors are triggered by the clock, so when the clock rises, data can pass through the latch. When the clock falls, the data is stored. For the negative edge triggered latch, the inputs to the pass transistors are switched.

The schematic for the Muller C-Element is shown in Figure 4.3. By feeding the output back into the series of gates, the C-Element is able to remember its past state.

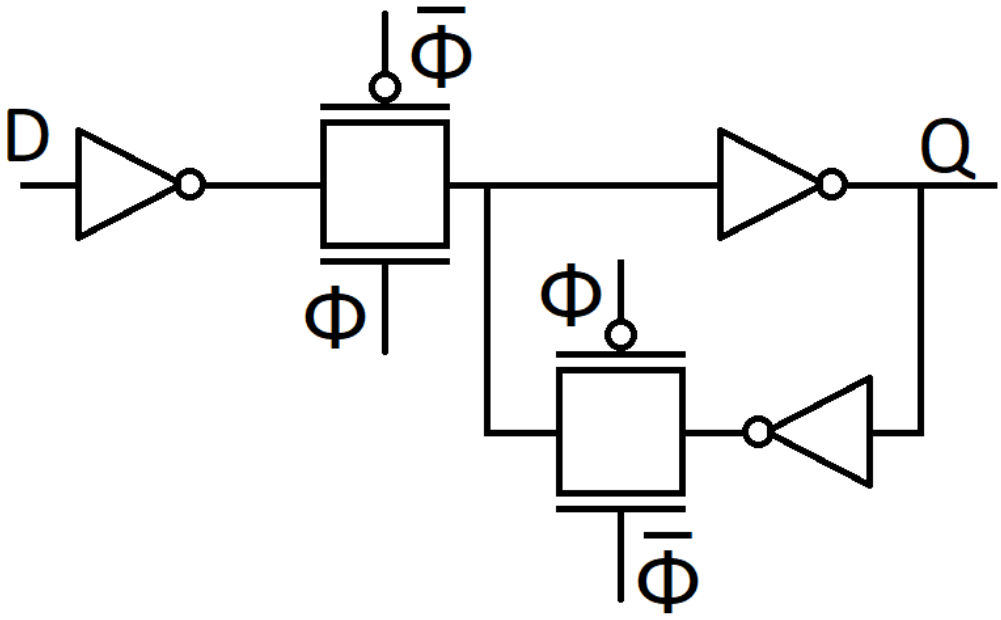

Figure 4.2: Latch schematic

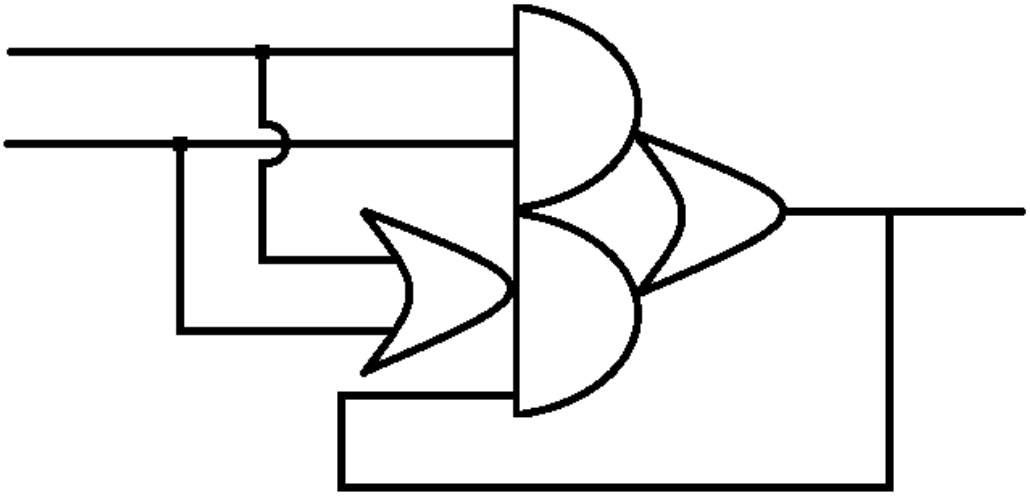

Figure 4.3: C-Element schematic

Gate	Width [*um*]
INV	0.6
NAND	0.8
NOR	1.2
AND	1.2
OR	1.6
C-Element	5.4
Latch	2.2
Latch_n	2.2
ISP Stage	10.6

Table 4.2: Widths of gates

Table 4.2 lists the width of all the gates. The height for every gate is 1.71 um.

Chapter 5

Layout Design

The following sections describe the modifications needed to turn the pipeline design into each type of pipeline and how each pipeline was laid out.

5.1 Asynchronous Pipeline

For the asynchronous pipeline, handshaking logic was added to the pipeline. The handshaking logic was shown in Figure 3.3.

The main component is the Muller C-Element. The C-Element uses the request signal from the previous stage and the acknowledge signal from the next stage to generate the enable for the latch, as well that stage's request and acknowledge signals.

The stall is initially taken from the input to the last latch for bit D. This is NANDed with the STALL_CLR signal and then ORed with the Reset signal to create the STALL_EN signal. The STALL_EN signal is ANDed with the output of the C-Elements for stage 3. The resulting signal is used to clock the stage 3 latches. This series of gates will disable the latch enable signal when there is a stall. To clear the stall, the STALL_CLR signal is set low.

If the system is in reset, the latches will automatically be enabled. This is done because the latch enable signal is also the request and acknowledge signals sent to the neighboring latches. Those signals need to be set high at the beginning of the simulation in order for the pipeline to run.

The asynchronous pipeline required additional logic for resetting the circuit. The request and acknowledge signals needed to be set high at the beginning of the simulation to ensure that data could move through the pipeline. Therefore, an OR gate was added to each of those signals, aside from the ones that were inputs into the pipeline, to force the signals high during reset. After reset, the data moving through the pipeline keeps them high as needed.

5.1.1 Layout

The layout for the asynchronous pipeline was designed with two goals in mind: keep the four gates for each set of handshaking logic close to their respective latches and place the cells so the data flows properly from one side to the other. To meet the latter goal, each stage of the pipeline was given its own row, with the input side on the left and the final output on the right. This would create four rows total, with each row following the format of latch-handshaking logic-combinational logic, repeated four times, with the stall logic at the end of the row. One row of this configuration is shown in Figure 5.1. However, having all the cells in one row would create a row about 50 um long, which is twice as long as the layouts for the other two pipelines. Additionally, there would be a high likelihood of congestion across the row because the handshaking signals would have to cross over all the other cells.

In Figure 5.1, LG stands for logic gate. Two of the four logic gates in each group are for ORing the acknowledge and valid signals with the reset signal. Another gate is the inverter for the acknowledge. The final gate is the logic gate associated with that stage.

To decrease the length of the layout, each stage was split into two rows. The top row would

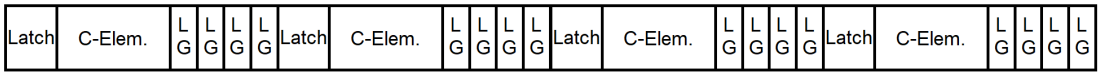

Figure 5.1: One row asynchronous layout

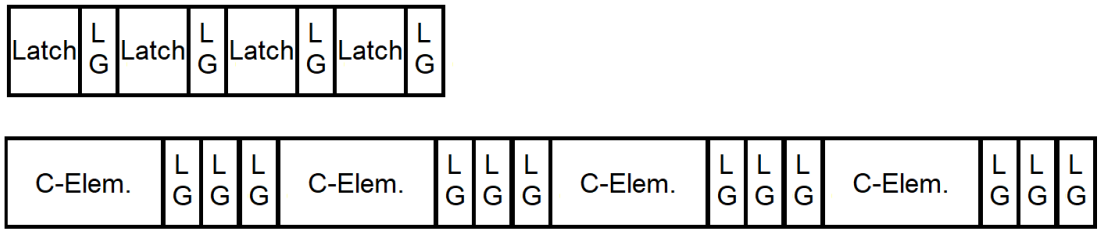

Figure 5.2: Two row asynchronous layout (unbalanced)

follow a latch-combinational pattern, while the bottom row would contain the handshaking logic. This would keep the handshaking logic close to the latches, while allowing the data to flow easily across the top row. While this layout plan meets both of the desired goals, the rows ended up very unbalanced in terms of length, with the top row being 12.6 um long and the bottom row being 36.8 um long.

To balance the two rows, a portion of the handshaking logic was moved into the top row. This put both rows at around 27 um. The final layout is shown in Figure 5.3.

For clarity, every other stage is colored gray. The logic gate to the left of each latch is the OR for the valid signal. The logic gate to the right of the latch is the logic gate for that stage. The last logic gate in the group is the OR for the acknowledge signal. The logic gates were placed in this way to minimize routing. The valid signal comes from the left, so it is placed on the left. The acknowledge signal comes from the right, so it is placed on the right. The logic gate for the data needs the output from the latch, so it is placed right next to it.

On the second row, the C-Elements are placed below their respective latches. The logic

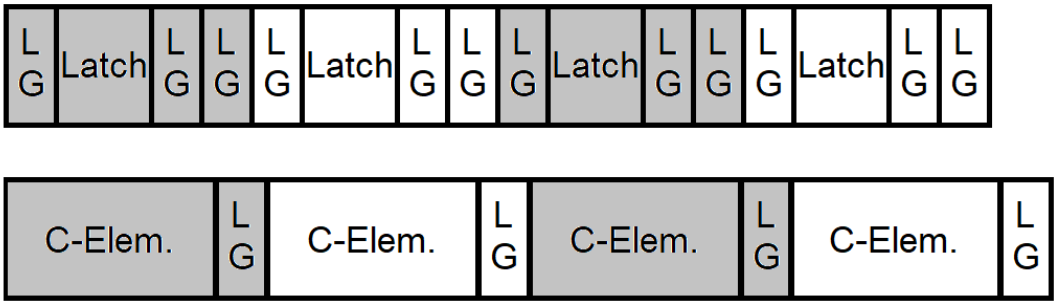

Figure 5.3: Two row asynchronous layout (balanced)

gate to the right is the inverter for the acknowledge. This layout places the signals close to the originating latches. The farthest required routing path is from the output of the data logic gate to the latch in the next latch row, below the C-Element row. There is no easy way to minimize that routing without compromising other paths.

Once the gates were laid out in this fashion, VDD and VSS rails were added to left and right side of the layout, respectively. Ideally, the input pins to the pipeline would be on the left and the output pins would be on the right, to match the data flow direction. Since the power rails were in the way, the pins had to be placed on the top and bottom. To minimize routing as much as possible, the pins for the first two stages were placed on the top and the pins for the second two stages were placed on the bottom. With this layout, the maximum distance a signal had to travel from latch to pin would be 3 rows. The final layout is shown in Figure 5.4.

The top row is a row with latches. The row below it has the C-Elements. This pattern alternates for the next six rows. The extra space on the right side was due to incorrect cell width measurements, which led to incorrect row width calculations. This does not affect the routing, as only VSS has to travel to that right side.

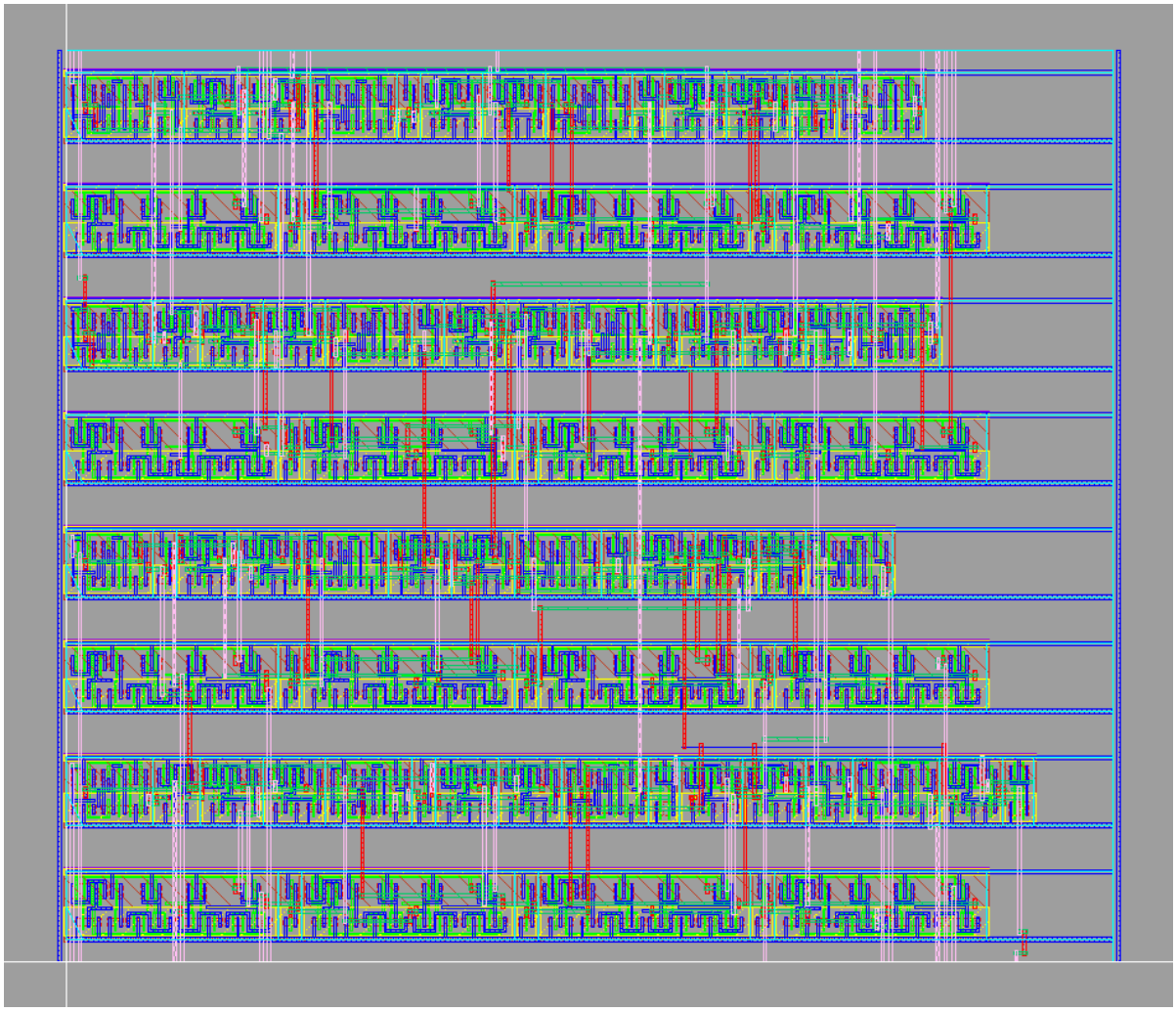

Figure 5.4: Layout of the Asynchronous Pipeline

5.2 Synchronous Pipeline

The synchronous pipeline is a 2-phase clocked pipeline. One clock signal is used to enable and disable every latch, and each stage of latches is clocked on the opposite edge of the clock. Instead of inverting the clock input for half the latches, the negative edge triggered latches are built to become transparent when the clock is low. This saves area, as no inverters or routing is needed to carry an inverted clock signal across the design.

To stall the pipeline, the stall signal is first NANDed with the STALL_CLR signal. This creates the STALL_EN signal, which is then ANDed with the global clock. If there is a stall, the clock is turned off and no latches will be triggered. To clear a stall, the STALL_CLR signal is set low.

As mentioned in Chapter 3, synchronous pipelines waste a large amount of power by clocking idle latches. The common solution is clock gating, which is used to shut down the idle parts of the circuit. Another issue with synchronous pipelines is stall boundaries. Stall boundaries are when the stall signal cannot travel to the preceding stages fast enough to stall them before the clock cycle finishes. This ends up with some data being lost, as the latch the data wants to move into is stalled, but the latch it is currently in clocks in new data [11].

Both of these problems contribute to the need for the interlocked synchronous pipeline. Therefore, it would be useful to create a circuit where these issues occur, to allow for comparison with the ISP. However, these problems require large circuits, and it would have been impractical to create such a circuit.

5.2.1 Layout

The synchronous pipeline was designed similarly to the asynchronous pipeline, with the goal of placing the cells to help the data flow from one side to the other. Again, the cells were split

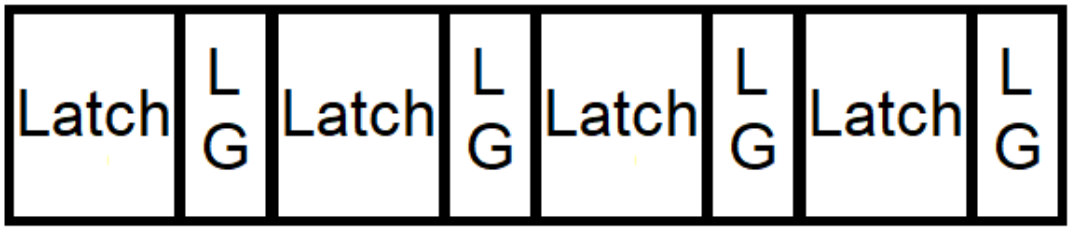

Figure 5.5: One row synchronous layout

into rows, one for each stage. Without the handshaking logic, only 4 rows were needed. Each row followed the pattern of latch-combinational logic. The stall logic was placed at the right end of the rows, closest to where the stall signal is generated. This put each row at around 13 um. One row of the designed layout is shown in Figure 5.5, with the other three rows looking similar.

Unlike the asynchronous pipeline, the cells for the synchronous pipeline were not manually placed. Instead, the router tool was used, to see if the router would come up with a similar design. While the designed layout makes sense from a human perspective, there was a chance that the router would find a more optimal layout. Figure 5.6 shows the router's layout.

The main difference between the designed layout and the router's layout is the placement of the latches. The design placed four latches in each row to spread them evenly across the layout. The router placed three latches in the top row, four in the second row, five in the third row, and four in the last row. The missing latch in the top row may have been moved because the STALL_CLR and CLK pins were placed at the top of the design. The NAND/AND gates for the stall logic with those signals may have taken the place of that fourth latch.

The pins were placed so that all of the input pins were placed at the top and all of the output pins were placed at the bottom. The ideal data flow would then be from top to bottom. The router did not follow this pattern. If it had, the top row would have three latches from the first stage. Instead, the top row has two latches from the first stage, and placed the other two at the

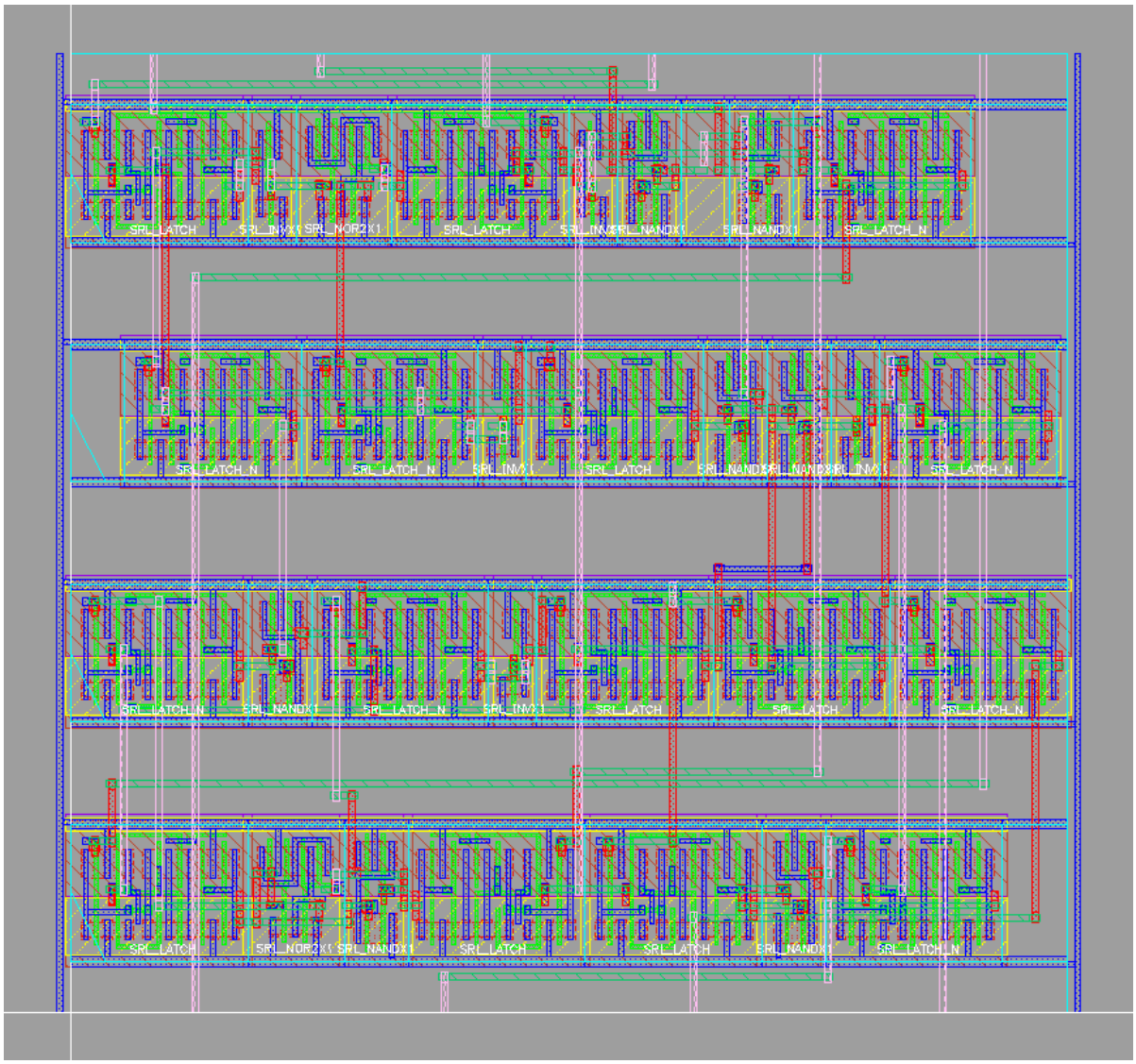

Figure 5.6: Layout of the Synchronous Pipeline

bottom of the design. This means that the input data has to travel from the top of the design all the way to the bottom row to reach those two latches. Placements like this increased the routing, so it would have been better to place the cells as they were in the designed layout.

5.3 Interlocked Synchronous Pipeline

The interlocked synchronous pipeline is based off the synchronous design with its global clock network. Each latch is augmented by the addition of two other latches: one for the valid bit and one for the stall bit. The stall latches are clocked on opposite edges of the global clock signal, as with the synchronous pipeline. The output of the stall latch is then combined with the clock signal to latch the valid latch. This results in the valid latch only getting clocked when there isn't a stall. The clock for the valid latch and the input to the valid latch are then combined to clock the data latch. This setup is arranged so the data latch is clocked only when the data is valid and there isn't a stall. The last piece of logic in the latch circuitry is that the input to the stall latch is ANDed with the output of the valid latch. This clears the stall if the data is invalid, allowing the data latch to overwrite the invalid data with valid data while the rest of the pipeline is stalled. The ISP latch circuitry was shown in Figure 3.6

For the ISP, the stall is ANDed with the STALL_CLR signal. It is then ORed with the stall signal coming from the stage 4 latches so it can override the stall handshaking signals. To clear the stall, STALL_CLR is set low.

5.3.1 Layout

The latch circuitry consists of three latches, two ANDs, and one NOR. In order to keep these elements together, the six gates were routed together into a "latch stage".

The pipeline was designed to flow from top to bottom to optimize the pin placements.

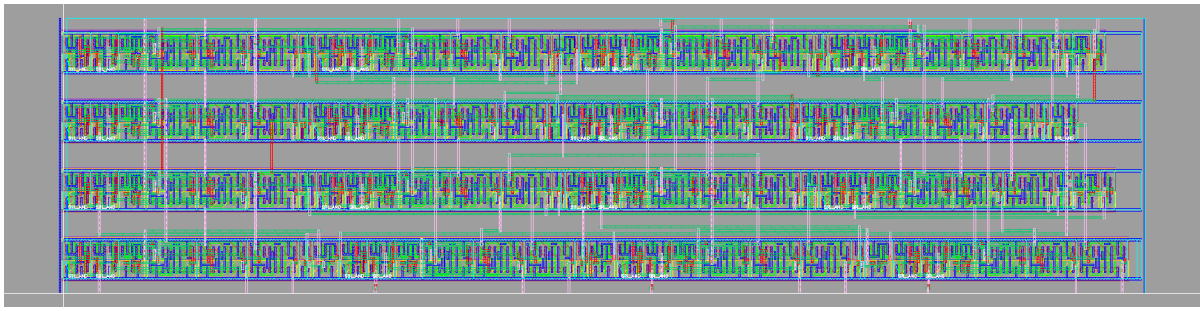

Figure 5.7: Layout of the Interlocked Synchronous Pipeline

Each row of the layout consists of the four latches from one stage along with the logic gate that follows each latch. The logic gates are placed to the right of their respective latches, similar to the original synchronous layout design. As all of the rows are ordered the same way, all the latches of each bit are placed in a roughly vertical stack. This minimizes the routing.

The final layout has four rows, each around 47 um wide, and is a total of 12 um tall. The height is the same as the synchronous pipeline, as the number of rows is the same, but the width is longer. The extra width comes from the additional logic in the latch circuitry.

Chapter 6

Results

6.1 Functionality & Timing

The following sections show in detail the operation of each pipeline. All three pipelines use the same series of inputs so that they can be fairly compared. The input sequence is shown in Table 6.1.

6.1.1 Asynchronous Pipeline

Figure 6.1 shows the output waveform for the asynchronous pipeline. Signals Ain through Din are the inputs for bits A through D. Signals Aout through Bout are the output signals from bits A through D. Signal D4d is the stall signal taken from the pipeline logic. STALL_CLR is the signal used to clear the stall. STALL_EN is the final stall signal after STALL_CLR has been taken into account. When STALL_EN is low, the pipeline is stalled. The last signal is the Reset signal. A period of reset is required to set all the handshaking signals high before pipeline operation so that data can move through it.

The five black rectangles mark where each of the inputs are entered into the circuit. The

Input Number	Input [hex]	Expected Output [hex]	Stall	Notes
1	2	8	0	–
2	D	0	0	–
3	8	9	0	This input creates a stall for the next cycle
4	F	9	1	The output should not change due to the stall and should continue to be 9.
5	F	0	0	The stall is cleared. The output is allowed to change.

Table 6.1: Input sequence for all three pipelines

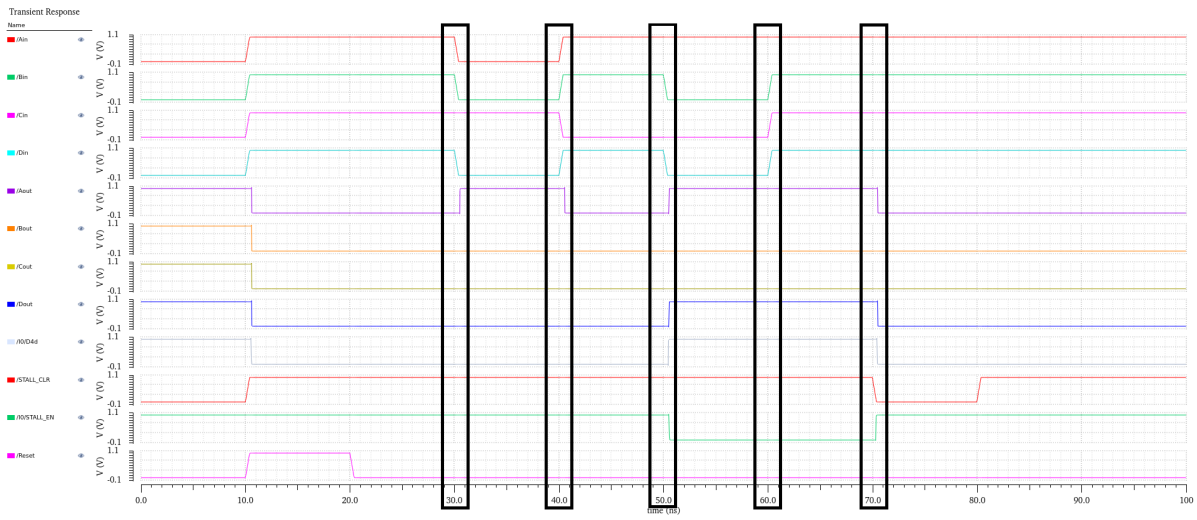

Figure 6.1: Waveform for the asynchronous pipeline

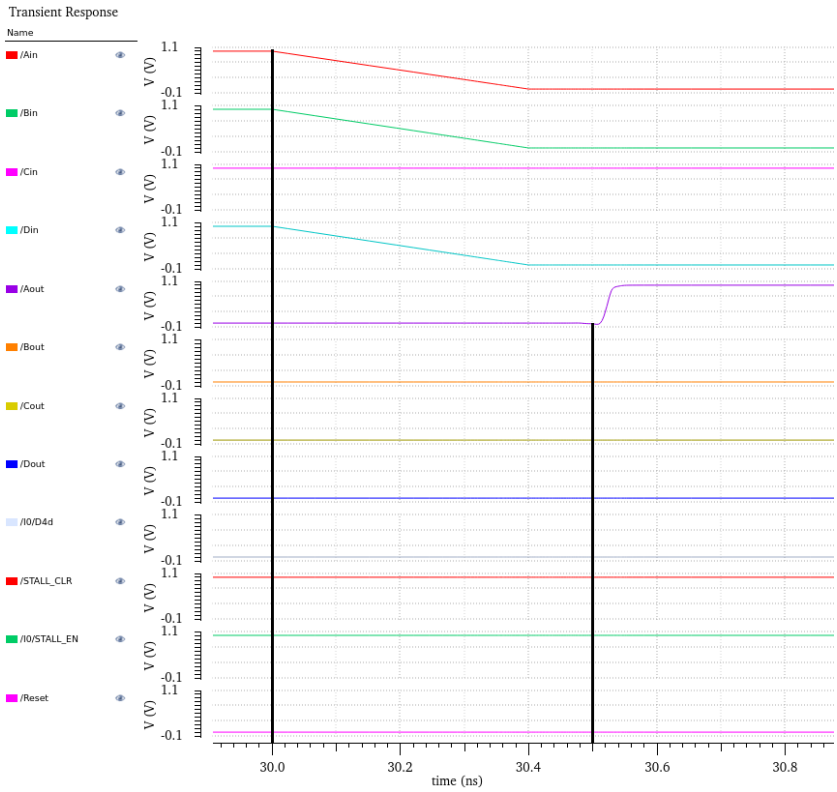

Figure 6.2: Close up of input/output transition for the asynchronous pipeline

inputs are entered every 10 ns. This is done to match the asynchronous pipeline to the synchronous and ISP pipelines, which run off a 200 MHz clock. A stall occurs in column 3. This stall prevents data from traveling through the pipeline, which can be seen in column 4, where the input changed to F but the output did not change to 0. STALL_CLR is deasserted in column 5, and STALL_EN returns high shortly after. This allows data to resume moving through the pipeline, and the output switches to 0.

Data takes 500 us to move through the pipeline, as shown by Figure 6.2.

Figure 6.3 shows data moving through all four stages for bit A. The first four signals are the values at each latch. The next two signals, R2a and A2a, are the request and acknowledge signals for stage 2. R3a and A3a are the request and acknowledge signals for stage 3. L3a is the latch enable for stage 3. STALL_EN, STALL_CLR, and Reset are the same as in the

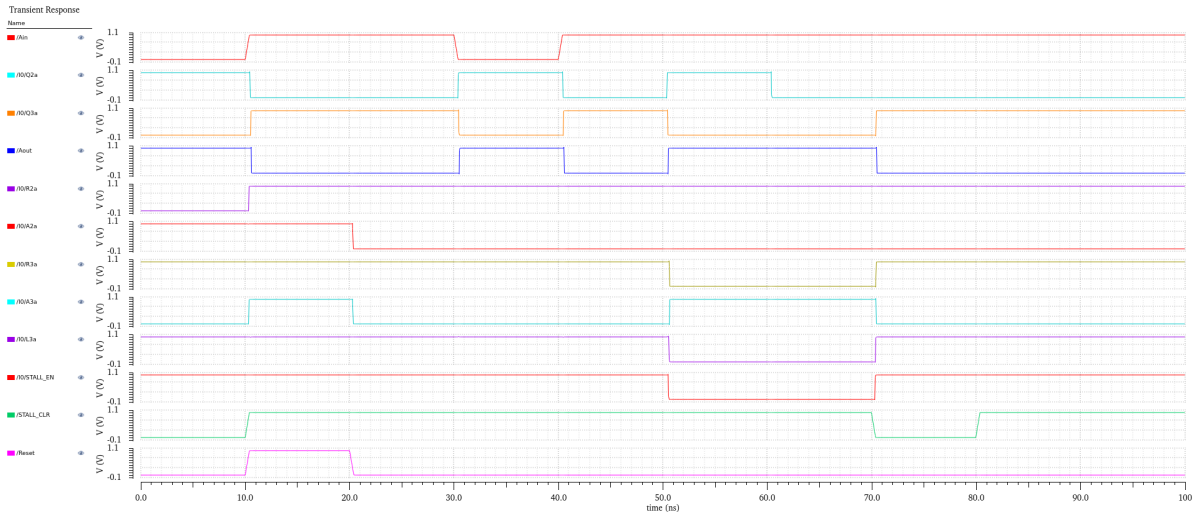

Figure 6.3: Data moving through bit A in the asynchronous pipeline

previous graphs.

The request and acknowledge signals start off as random values at the beginning of the simulation because they are internal signals and therefore not set to any value. After reset, they are set to their proper values: high for request and low for acknowledge. The acknowledge signal is inverted before it reaches the C-Element, so a low value here means a high value for the C-Element. The request and acknowledge signals stay at their respective values throughout the simulation, allowing data to pass through the pipeline, until the stall occurs. The stall starts at roughly 50.5 ns, when STALL_EN goes low. This causes L3a to go low, which in turn causes R3a to go low and A3a to go high. Latch 3 is now disabled and has told its neighbors that it is not accepting new data.

The value of Q2a goes low at the 60 ns mark. This is due to Bin changing value, which was shown in Figure 6.1. This change does not propagate to latch 3 because latch 3 is stalled. When the stall is cleared at 70 ns, the data can continue moving and latch 3 is allowed to update its value.

The input data travels through the pipeline as expected, with each latch switching shortly

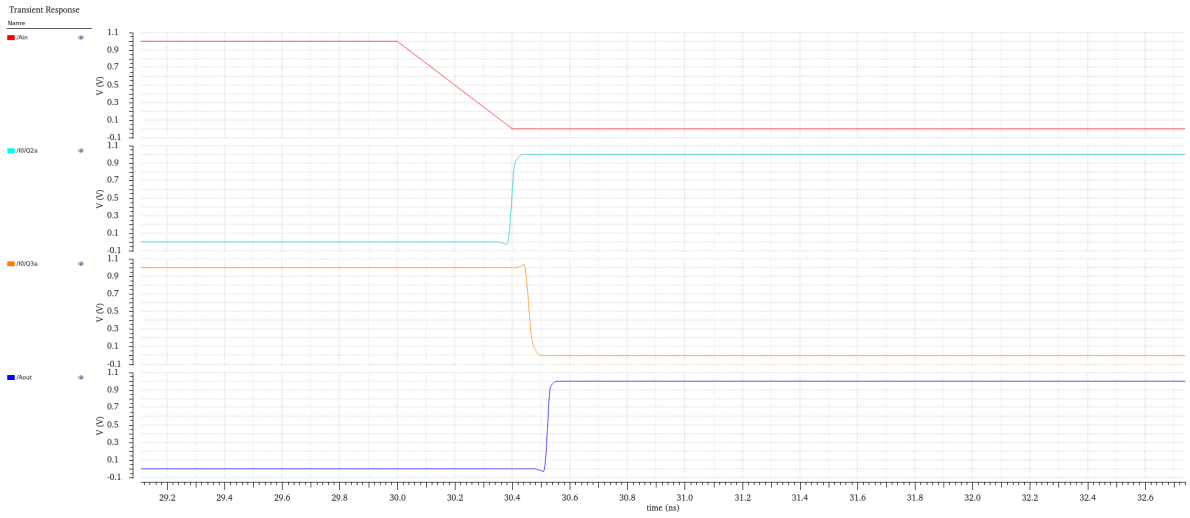

Figure 6.4: Data moving through latches of bit A in the asynchronous pipeline

after the one before it. This can be seen in more detail in Figure 6.4.

Ain has the longest transition time because the change is generated from the test bench, while the other signals are latches changing value. The latches take around 100 us to switch. The latches begin to switch when the previous latch is at roughly 50% VDD, or 0.5 V.

6.1.2 Synchronous Pipeline

The waveform for the synchronous pipeline is shown in Figure 6.5. The first four signals, Ain through Din, are the inputs. The next four signals, Aout through Dout, are the outputs. G_CLK is the global clock. This clock is generated by the test bench. The next signal, CLK, is the clock that is used to clock all the latches. This clock is a combination of G_CLK and the stall condition. The last three signals are for the stall. D4_Stall is the stall signal generated by the pipeline (called D4d in the asynchronous pipeline), STALL_CLR is the clear signal, and EN is the STALL_EN signal. As there are no internal handshaking signals, the pipeline does not need to be reset, and therefore there is no Reset signal.

The synchronous pipeline runs off the global clock signal, with new inputs introduced at

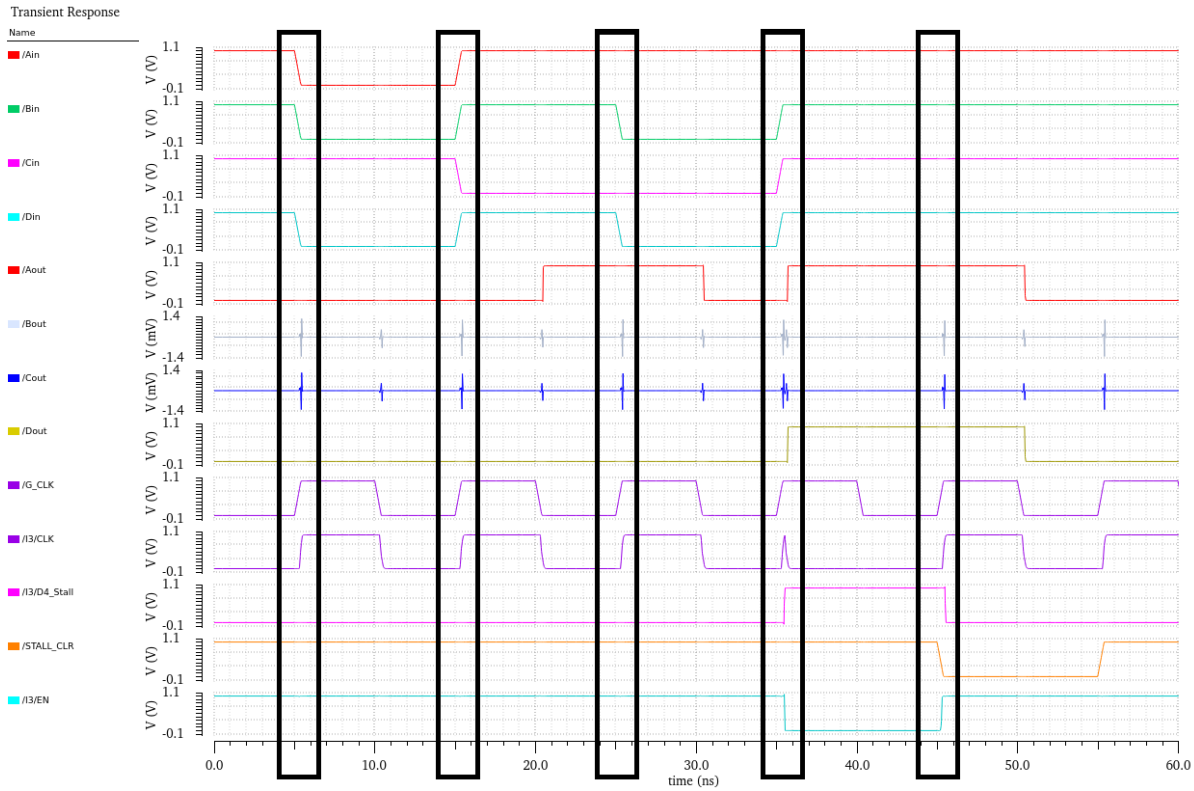

Figure 6.5: Waveform for the synchronous pipeline

each rising edge. The inputs follow the same pattern as described in Table 6.1. As mentioned in Chapter 4, the pipeline design only gives five different outputs for the full range of 16 inputs, and of those five only one output sets Bout and Cout high. This leads to a problem in the synchronous pipeline: since Bout and Cout never change, they stay at 0 V for the entire simulation. The noisiness in their waveform is related to this. In simulations where Bout and Cout can become high, there is no noisiness when they are low.

Aside from the noisiness, the pipeline operates correctly. The inputs are entered on the rising edge of the clock and appear at the output 1.5 cycles later, at the falling edge of the clock. When the stall occurs, data stops moving through the pipeline. The stall starts at 35 ns, in the fourth rectangle. The stall condition forces CLK low. Without the switching of the clock, the latches can no longer activate. When STALL_CLR is deasserted at 45 ns, the stall is cleared, and CLK resumes operation. Data moves through the pipeline again and then shows up at the output at around 50 ns, half a clock cycle after the stall was cleared.

The transfer of data between latches is shown in Figure 6.6.

Data moves through the pipeline with every clock edge. Ain falls at 5 ns with the positive clock edge. QA2, the signal from the second latch, rises at 10 ns with the negative edge. QA3 falls with the positive edge, and Aout rises with the negative edge. It therefore takes 1.5 clock cycles for the change in Ain to propagate to Aout. The stall can be seen at the 40 ns mark. No data changes with the negative edge.

6.1.3 Interlocked Synchronous Pipeline

The interlocked synchronous pipeline (ISP) operates very similarly to the synchronous pipeline. Figure 6.7 shows the waveforms for the ISP. Ain through Din are the inputs, Aout through Dout are the outputs, CLK is the global clock, and D4d, STALL_CLR, and STALL_EN are the stall signals.

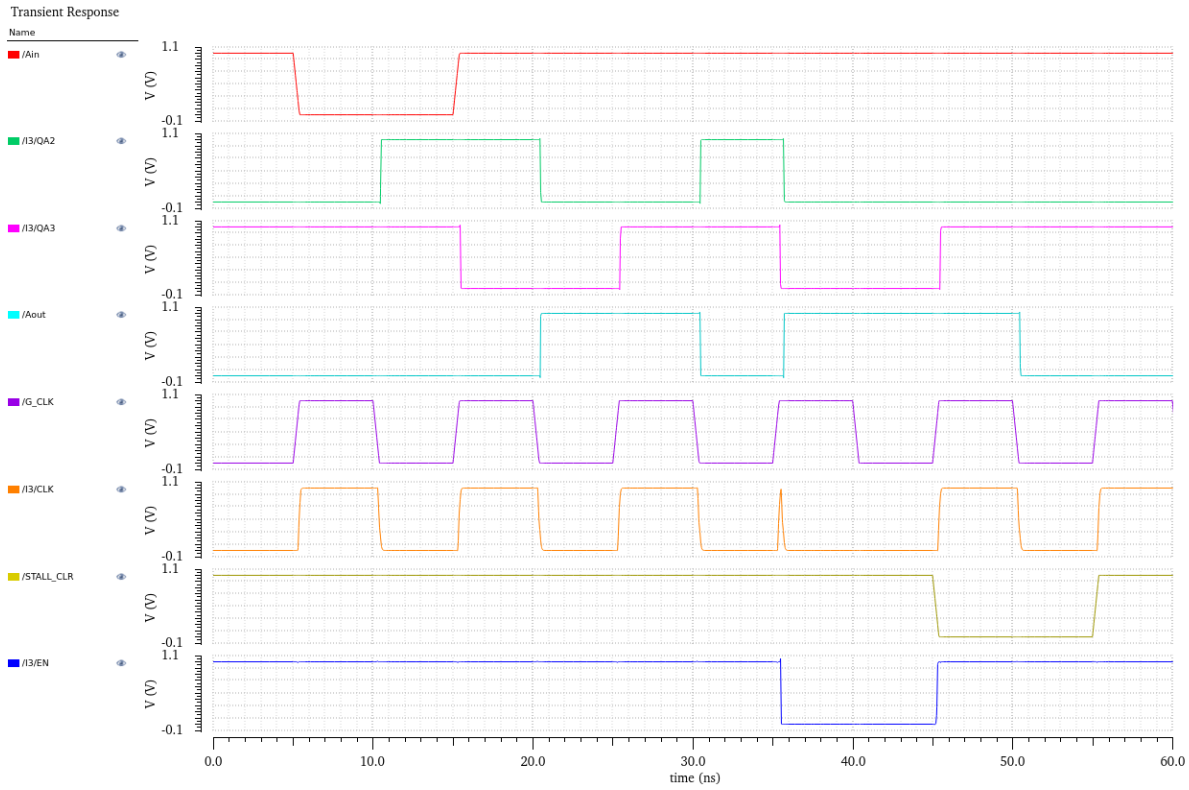

Figure 6.6: Data moving through bit A in the synchronous pipeline

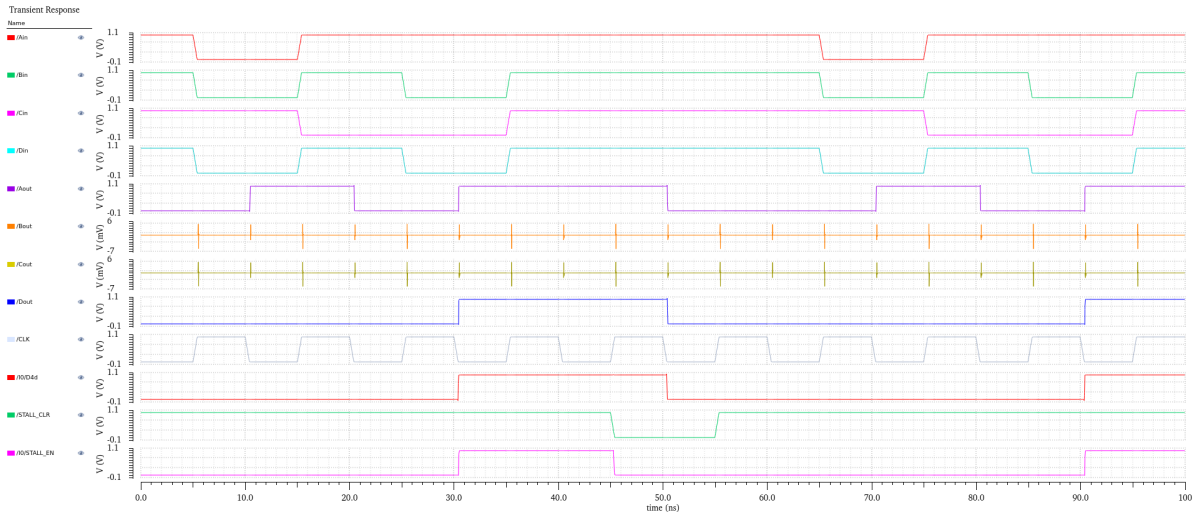

Figure 6.7: Waveform for the interlocked synchronous pipeline

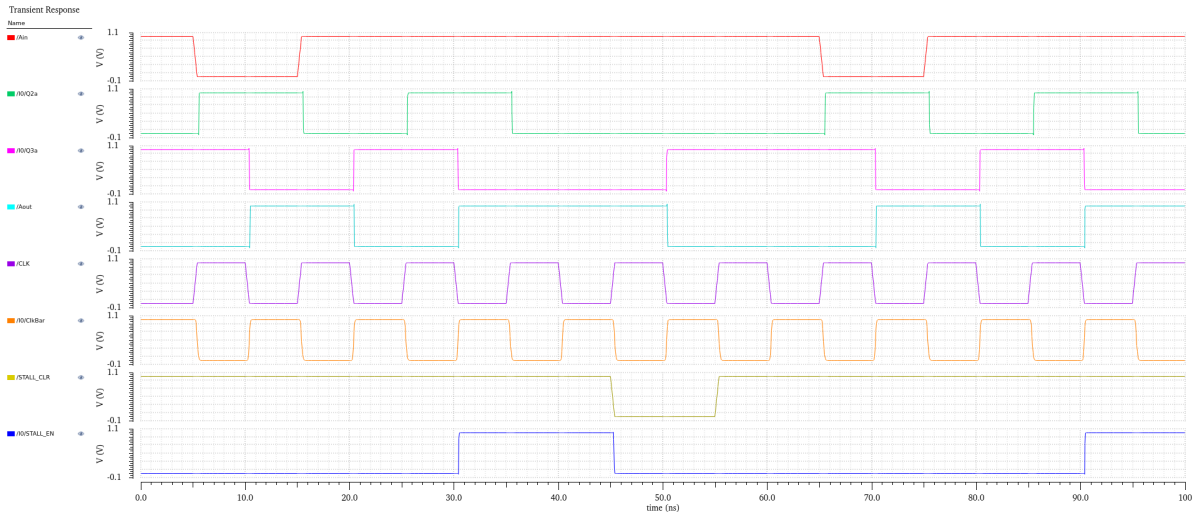

Figure 6.8: Data moving through bit A in the ISP

The inputs for the ISP are the same as for the asynchronous and synchronous pipelines. The same problem with Bout and Cout being noisy occurs as it did in the synchronous pipeline. The major difference between the ISP and the synchronous pipeline is how long it takes data to move through the pipeline. The synchronous pipeline takes 1.5 clock cycles. The ISP takes only 0.5 clock cycles. The reason for this can be seen in Figure 6.8.

Like the synchronous pipeline, the latches of the ISP are clocked on alternating clock edges. However, while the synchronous pipeline uses latches built to use the negative edge of the clock, the ISP uses an inverted clock signal for those latches. At 10 ns, the rising edge of ClkBar triggers latch 3. The data travels through the logic and arrives at latch 4. The expected operation is that the data then waits at latch 4 until latch 4 is clocked at the next negative edge of the clock, at 20 ns. The latches are not triggered by the change in the clock signal, though, but by the value of the clock being low. At 10 ns the CLK signal falls, making latch 4 transparent to new data for the next 5 ns. The new data from latch 3 arrives at latch 4 and is then picked up immediately. The end result is that latch 3 and latch 4 are both transparent from 10 ns to 15 ns. The same principal occurs with latches 1 and 2, which causes the latency

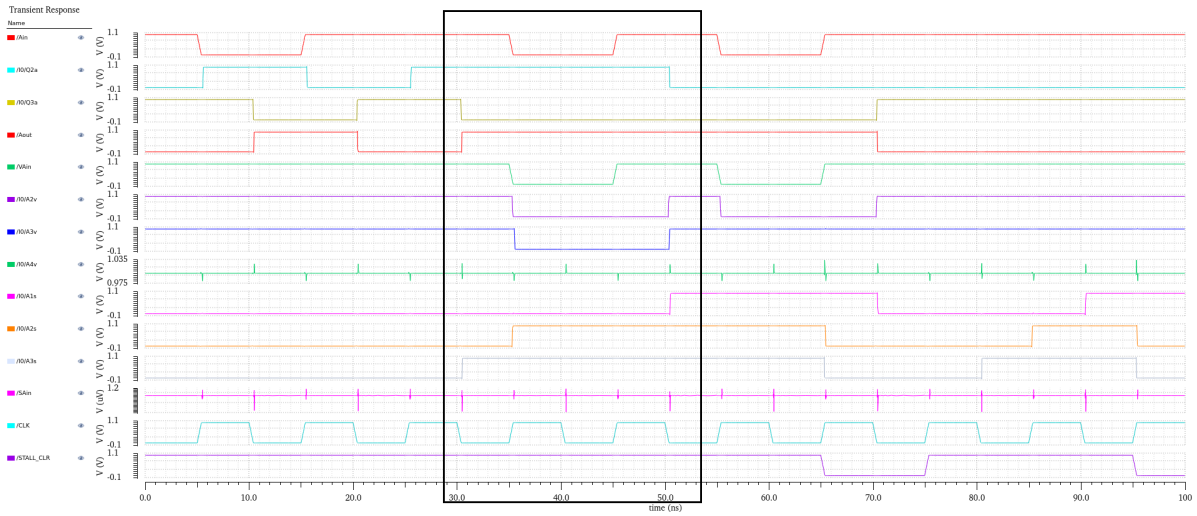

Figure 6.9: Data, valid, and stall signals for bit A for the ISP

to be only 0.5 cycles.

Figure 6.9 shows the ISP operation with the valid and stall signals for bit A. The first four signals are the four latches for bit A. VAin is the valid signal for the first latch, generated by the test bench, and signals A2v through A4v are valid signals for the other stages. A1s through A3s are the stall signals for stages 1 to 3. SAin is the stall for the last stage, which is generated by the test bench. Finally there are the CLK and the STALL_CLR signals. SAin is always low because the stall occurs before stage 4 and only affects stages 1 through 3. A4v is always high because the inputs are set up so that the stall clears out all invalid data before it reaches stage 4.

The area of interest for this waveform is captured within the box. At 30 ns, the stall signal (not shown) goes high. The stall for stage 3, A3s, immediately goes high because of this. At the next positive edge of the clock, the stall for stage 2, A2s, is updated and goes high. The stall for stage 1 does not go high until a cycle later. This is because of the invalid data in the pipeline. At 35 ns, Ain is given the value of 0. This data is marked invalid by VAin. When the stall reaches the invalid data at stage 1, they cancel each other out. The valid data entered at

Pipeline	Width [*um*]	Height [*um*]	Area [um^2]
Asynchronous	27	23.14	624.78
Synchronous	12.75	11.94	152.235
ISP	47.15	11.94	562.971

Table 6.2: Areas of the three pipelines

45 ns can move into stage 1 because the stall is still low. This overwrites the invalid data. Now that the data is valid, the stall goes high to prevent the valid data from being overwritten. This can be seen at 50 ns, where A2v and A1s both go high.

This setup is analogous to Figure 3.7. Because of the invalid data, the length of the stall is shortened for stage 1. Stage 3 has a stall of 35 ns (3.5 clock cycles) while stage 1 is only stalled for 20 ns (2 clock cycles).

6.2 Area

The areas of the three pipelines are shown in Table 6.2.

The synchronous pipeline is the clear winner in terms of area. The asynchronous pipeline is larger because of the extra handshaking logic, while the ISP is larger because it has two extra latches per stage. These numbers show that despite the ISP having three times the number of latches, it is still smaller than the asynchronous pipeline. However, the asynchronous pipeline has eight rows of cells instead of four, like the synchronous pipeline and ISP do. The extra space between the additional rows makes the asynchronous pipeline appear larger than it actually is. A more fair comparison can be done by looking at the gates making up each pipeline. Table 6.3 shows the breakdown of each pipeline in terms of its gates. For this comparison, the ISP stage (the three latches plus the latch circuitry) is broken down into its individual gates.

The asynchronous pipeline has extra inverters due to inverting the acknowledge signal. The extra OR gates are for resetting the handshaking signals. The ISP's extra gates comes from the

Gate	Asynchronous	Synchronous	ISP
INV	16	5	9
NAND	6	7	5
NOR	2	2	22
AND	4	0	33
OR	25	0	0
C-Element	16	0	0
Latches	16	16	48
Total	85	30	117

Table 6.3: Number and types of gates in each pipeline

Pipeline	Total Area [um^2]	Gate Area [um^2]
Asynchronous	624.78	313.272
Synchronous	152.235	79.002
ISP	562.971	309.51

Table 6.4: Areas of the three pipelines in terms of gates only

2 additional latches, 2 AND gates, and 1 NOR gate for each of the 16 latches. The synchronous pipeline has only three extra gates: two NANDs and an inverter for the stall logic.

Using these gate counts and the gate dimensions from Table 4.2, the areas of the pipelines are shown in Table 6.4.

The synchronous pipeline is still the smallest pipeline, at around 25% the size of the other two. The asynchronous pipeline is still larger than the ISP but the difference between them is now too small to be significant.

6.3 Latency & Throughput

The latency of each pipeline is how long it takes for the input data to affect the output signals. Typically this is measured in clock cycles. Using this method, the synchronous pipeline has a latency of 1.5 clock cycles and the ISP has a latency of 0.5 clock cycles. The asynchronous

pipeline does not have a clock and cannot be measured this way. The other method of measuring latency is to go by time units. The asynchronous pipeline has a latency of 500 us, as shown earlier. The synchronous pipeline and ISP have a latency of 15,000 us and 5,000 us, respectively. However, this is not a fair comparison. The data transfer for those two pipelines is limited by the clock frequency. If the clock was faster, the synchronous pipeline and ISP would operate closer to the speed of the asynchronous pipeline. They would still be hindered by the clock though and thus remain slightly slower than the asynchronous pipeline.

The throughput of a pipeline is how much data can move through it in a given time. Since the input data is entered every 10 ns, the maximum throughput possible is one output per clock cycle. This equalizes the asynchronous pipeline and the ISP for this type of comparison. The synchronous pipeline falls behind as it needs an extra half cycle to finish processing one input. The ISP has the additional feature of valid bits, however, which has the potential to increase its throughput. If invalid data goes through an asynchronous or synchronous pipeline, that data will end up at the output. There is no way to remove it. If a stall occurs in an ISP pipeline, the timing may work out so that the invalid data is canceled out by the stall and more valid data can fill the pipeline. The end result could be, for example, that the asynchronous and synchronous pipeline output 5 valid outputs and 2 invalid ones, while the ISP could output 7 valid ones. The throughput in terms of valid vs invalid data is dependent on the timing of the stalls and cannot be directly measured, but it must be noted as an occasional benefit of the ISP.

Chapter 7

Discussion

The goal of the interlocked synchronous pipeline (ISP) is to reduce power consumption by adding handshaking signals to synchronous pipelines. Unfortunately, power could not be measured with the development tools used. Instead, other metrics have to be looked at to judge the usefulness of the ISP.

The first metric is area. Area is directly related to power because the larger the area used, the more gates are in the circuit, and more gates will use more power. Compared to the synchronous pipeline, the ISP is 3.7 times larger. In terms of cell area only, the ISP is 3.9 times larger. The increase is due to the extra latches. The ISP is closer in size to the asynchronous pipeline, but this comparison is not important. Designers are highly unlikely to switch from asynchronous pipelines to an ISP as there are no real benefits moving in this direction. While there are benefits moving from the synchronous pipeline to the ISP, the large area increase will put many designers off. Larger area not only leads to more power, but also requires a larger die. A larger die size means that the cost of making the chip will increase, on top of the cost for the extra design work needed.

In terms of timing, the asynchronous pipeline performed the best. This is to be expected.

One of the main advantages of the asynchronous pipeline is its speed, and this work proved that this is true. Between the ISP and the synchronous pipeline, the ISP performed better. However, this may be a flaw in the design. Since the latches are driven by the value of the clock and not the edge itself, having a negative version of the clock led to two stages operating at once in the ISP. The synchronous pipeline used latches built for the negative edge of the clock so only one clock signal was needed, which avoided this problem. If the two pipelines had used the same clock method, either both having a negative version of the clock or both using negative edge latches and one clock, or if the latches had been designed to be properly edge triggered, they likely would have had the same speed.

The main attraction of the ISP is its ability to deal with stalls and invalid data. Stalls occur very frequently in digital circuits, especially in processors that have to deal with branch instructions and memory operations. Local stall signals that don't require long, timing critical signals, would make the circuit design simpler. Removing invalid data instead of stalling is also convenient as it allows more data to be processed. There is no way to do this with synchronous pipelines without adding additional circuitry to flush the data path.

The final point to consider is the ability to use synchronous tools to generate the ISP. One of the biggest hurdles in creating large asynchronous pipelines is that there exist very few development tools that can handle them. The ISP aims to make an asynchronous-like design that can be created with the common synchronous pipeline development tools. These development tools were not used for this project, and therefore there is no concrete data that shows if the ISP can or cannot be used with them. However, in theory it should work. From an RTL standpoint, a designer only has to replace the latch or flop definition with an ISP stage. The valid and stall signals can be wired the same way the rest of the design is. From a physical design standpoint, the designer will need a standard ISP stage cell. Letting a synthesis tool place all the gates required for the ISP stage manually will likely result in poorly optimized placements. Creating

a standard cell that routes all the gates together will work more effectively, but requires that a designer spends the time to build such a cell.

To summarize, the advantages of the ISP are that it has locally generated stalls, the ability to remove invalid data during stalls to increase throughput, and a potentially faster cycle time. The disadvantages are that it requires extra work to implement and the area is close to 4 times the size of the synchronous pipeline. Compared to the asynchronous pipeline, the ISP has around the same area and is easier to implement, but it operates slower.

Whether or not the ISP is a worthwhile alternative to synchronous pipelines is based on the design. A digital circuit that only cares about throughput may like the ISP for its invalid data removal and not care about the area. On the other hand, a microprocessor could not handle the area increase. Power is also an important factor, and the expected power increase would likely dissuade most designers.

Chapter 8

Conclusion

The paper "Synchronous Interlocked Pipelines" by Jacobson et al. introduces the concept of an interlocked synchronous pipeline (ISP) as an alternative to synchronous pipelines. To test the effectiveness of this new pipeline, an asynchronous pipeline, synchronous pipeline, and ISP were built in a generic 45 nm library. All three pipelines were built off the same pipeline logic design so that they could be readily compared. The waveforms for each pipeline were analyzed to prove that the pipeline worked correctly and could handle stalls.

The pipelines were compared in terms of area, timing, and throughput. For area, the asynchronous and ISP were nearly four times the size of the synchronous one. For timing, the asynchronous pipeline processed data the fastest. This ended up being a poor metric because the clock speed of the synchronous pipeline and ISP could have been increased to allow them to match the asynchronous speed. In addition, the ISP's clock was designed differently from the synchronous pipeline's clock, which allowed it to perform faster than it should have. Ignoring timing differences, the ISP had the highest throughput. Due to its valid/stall handshaking signals, it has the potential to remove invalid data and allow valid data to continue to be processed during a stall. Under the right conditions, the ISP could remove all invalid data in

the pipeline and thus have a higher valid throughput than the asynchronous and synchronous pipelines, which have no way to clear invalid data.

Other metrics that could not be compared are power and the ease of creating the ISP with synchronous tools. The latter is theorized to be simple, though it would require extra design work to setup. The former could not be measured with the tools used. As power is an important metric to consider when designing a pipeline, the fact that it could not be measured severely limits the ability to determine which of the three pipelines is the best choice.

The choice of pipeline ultimately depends on the type of digital circuit being designed. For microprocessors, it would be better to stick to synchronous pipelines. The area increase, and likely power increase due to the extra gates, would not be acceptable. However, there are likely applications were the ISP's ability to remove invalid data makes the area increase worthwhile.

More work could be done with the three pipelines to determine the advantages and disadvantages. Recreating the pipelines with a tool that could measure power consumption would be the highest priority. Aside from that, the pipelines could also be modeled in RTL. The ease at which they could be coded and synthesized would prove whether or not the ISP could be used with synchronous design tools. Finally, larger circuits could be explored. With a four stage pipeline, some of the problems with synchronous pipelines that the ISP tries to fix could not occur. For example, the circuit is too small to experience stall boundaries, where the stall cannot propagate fast enough to stop the clock from clocking in new data. There are also no timing constraints on the data paths in the pipeline because of the small logic depth. Creating a larger digital circuit that experiences these issues could provide evidence of the ISP being a worthwhile choice.

References

[1] M. Kovac and M. Kubicek. Asynchronous Logical System Simulation in VHDL. In *2008 18th International Conference Radioelektronika*, pages 1–4, April 2008. doi:10.1109/RADIOELEK.2008.4542725.

[2] K. Gupta, N. Pandey, and M. Gupta. A novel active shunt-peaked MOS Current Mode Logic C-element for asynchronous pipelines. In *2011 International Conference on Multimedia, Signal Processing and Communication Technologies*, pages 122–125, Dec 2011. doi:10.1109/MSPCT.2011.6150453.

[3] Xiao Yong and Zhou Runde. Single-track asynchronous pipeline controller design. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, volume 2, pages 764–768 Vol. 2, Jan 2005. doi:10.1109/ASPDAC.2005.1466453.

[4] L. Manuel, C. K. Midhun, and R. K. Kavitha. High speed dynamic asynchronous pipeline: Self-Controlled approach. In *2012 Annual IEEE India Conference (INDICON)*, pages 592–596, Dec 2012. doi:10.1109/INDCON.2012.6420687.

[5] K. Sravani and R. Rao. High throughput and high capacity asynchronous pipeline using hybrid logic. In *2017 International Conference on Innovations in Electron-*

ics, Signal Processing and Communication (IESC), pages 11–15, April 2017. doi:10.1109/IESPC.2017.8071856.

[6] T. Werner and V. Akella. Asynchronous processor survey. *Computer*, 30(11):67–76, Nov 1997. doi:10.1109/2.634866.

[7] Je-Hoon Lee, Won-Chul Lee, and Kyoung-Rok Cho. A novel asynchronous pipeline architecture for CISC type embedded controller, A8051. In *The 2002 45th Midwest Symposium on Circuits and Systems, 2002. MWSCAS-2002.*, volume 2, pages II–II, Aug 2002. doi:10.1109/MWSCAS.2002.1186952.

[8] H. A. Farouk and M. T. El-Hadidi. Implementing Globally Asynchronous Locally Synchronous processor pipeline on commercial synchronous FPGAs. In *2010 17th International Conference on Telecommunications*, pages 989–994, April 2010. doi:10.1109/ICTEL.2010.5478856.

[9] D. L. Oliveira, T. Curtinhas, L. A. Faria, and L. Romano. Design of synchronous pipeline digital systems operating in double-edge of the clock. In *2013 IEEE 4th Latin American Symposium on Circuits and Systems (LASCAS)*, pages 1–4, Feb 2013. doi:10.1109/LASCAS.2013.6519068.

[10] A. Branover, R. Kol, and R. Ginosar. Asynchronous design by conversion: converting synchronous circuits into asynchronous ones. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 870–875 Vol.2, Feb 2004. doi:10.1109/DATE.2004.1268996.

[11] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers. Synchronous interlocked pipelines. In *Proceedings Eighth International*

Symposium on Asynchronous Circuits and Systems, pages 3–12, April 2002. doi:10.1109/ASYNC.2002.1000291.

[12] N. Saxena, S. Dutta, N. Pandey, and K. Gupta. Implementation of asynchronous pipeline using Transmission Gate logic. In *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, pages 101–106, March 2016. doi:10.1109/ICCTICT.2016.7514560.

[13] B. Su, L. Shen, L. Wang, Z. Wang, Y. Wang, L. Huang, and W. Shi. DCP: Improving the Throughput of Asynchronous Pipeline by Dual Control Path. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 230–237, Nov 2013. doi:10.1109/HPCC.and.EUC.2013.42.

[14] M. Gholipour, K. Shojaee, A. Khademzadeh, A. Afzali-Kusha, and M. Nourani. Performance and power analysis of asynchronous pipeline design methods. In *Proceedings. The 16th International Conference on Microelectronics, 2004. ICM 2004.*, pages 409–412, Dec 2004. doi:10.1109/ICM.2004.1434600.

[15] P. A. Beerel. Asynchronous circuits: an increasingly practical design solution. In *Proceedings International Symposium on Quality Electronic Design*, pages 367–372, March 2002. doi:10.1109/ISQED.2002.996774.

[16] C. K. Midhun, J. Joy, and R. K. Kavitha. High-Speed Dynamic Asynchronous Pipeline: Self-Precharging Style. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(10):2235–2239, Oct 2014. doi:10.1109/TVLSI.2013.2282834.

[17] M. Singh and S. M. Nowick. The Design of High-Performance Dynamic Asynchronous

Pipelines: High-Capacity Style. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(11):1270–1283, Nov 2007. doi:10.1109/TVLSI.2007.902206.

[18] O. Hauck and S. A. Huss. Asynchronous wave pipelines for high throughput datapaths. In *1998 IEEE International Conference on Electronics, Circuits and Systems. Surfing the Waves of Science and Technology (Cat. No.98EX196)*, volume 1, pages 283–286 vol.1, Sep. 1998. doi:10.1109/ICECS.1998.813322.

[19] P. Balaji, W. Mahmoud, E. Ososanya, and K. Thangarajan. Survey of the counterflow pipeline processor architectures. In *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory (Cat. No.02EX540)*, pages 1–5, March 2002. doi:10.1109/SSST.2002.1026993.

[20] P. G. Lucassen and J. T. Udding. A process-algebraic approach to the design of asynchronous (counterflow) pipelines. In *IEE Colloquium on Design and Test of Asynchronous Systems*, pages 7/1–7/6, Feb 1996. doi:10.1049/ic:19960252.

[21] J. Butas and J. Povazanec. A fine-grain asynchronous pipeline reaching the synchronous speed. In *ASICON 2001. 2001 4th International Conference on ASIC Proceedings (Cat. No.01TH8549)*, pages 547–550, Oct 2001. doi:10.1109/ICASIC.2001.982621.

[22] A. E. Sjogren and C. J. Myers. Interfacing synchronous and asynchronous modules within a high-speed pipeline. In *Proceedings Seventeenth Conference on Advanced Research in VLSI*, pages 47–61, Sep. 1997. doi:10.1109/ARVLSI.1997.634845.

[23] N. Toosizadeh, S. G. Zaky, and J. Zhu. VariPipe: Low-overhead variable-clock synchronous pipelines. In *2009 IEEE International Conference on Computer Design*, pages 117–124, Oct 2009. doi:10.1109/ICCD.2009.5413167.

[24] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 657–662, July 2006. doi:10.1145/1146909.1147077.

[25] A. Rettberg, M. Zanella, T. Lehmann, and C. Bobda. A new approach of a self-timed bit-serial synchronous pipeline architecture. In *14th IEEE International Workshop on Rapid Systems Prototyping, 2003. Proceedings.*, pages 71–77, June 2003. doi:10.1109/IWRSP.2003.1207032.

Appendix I

Schematics

This appendix contains the schematics for all of the gates and pipelines created for this project.

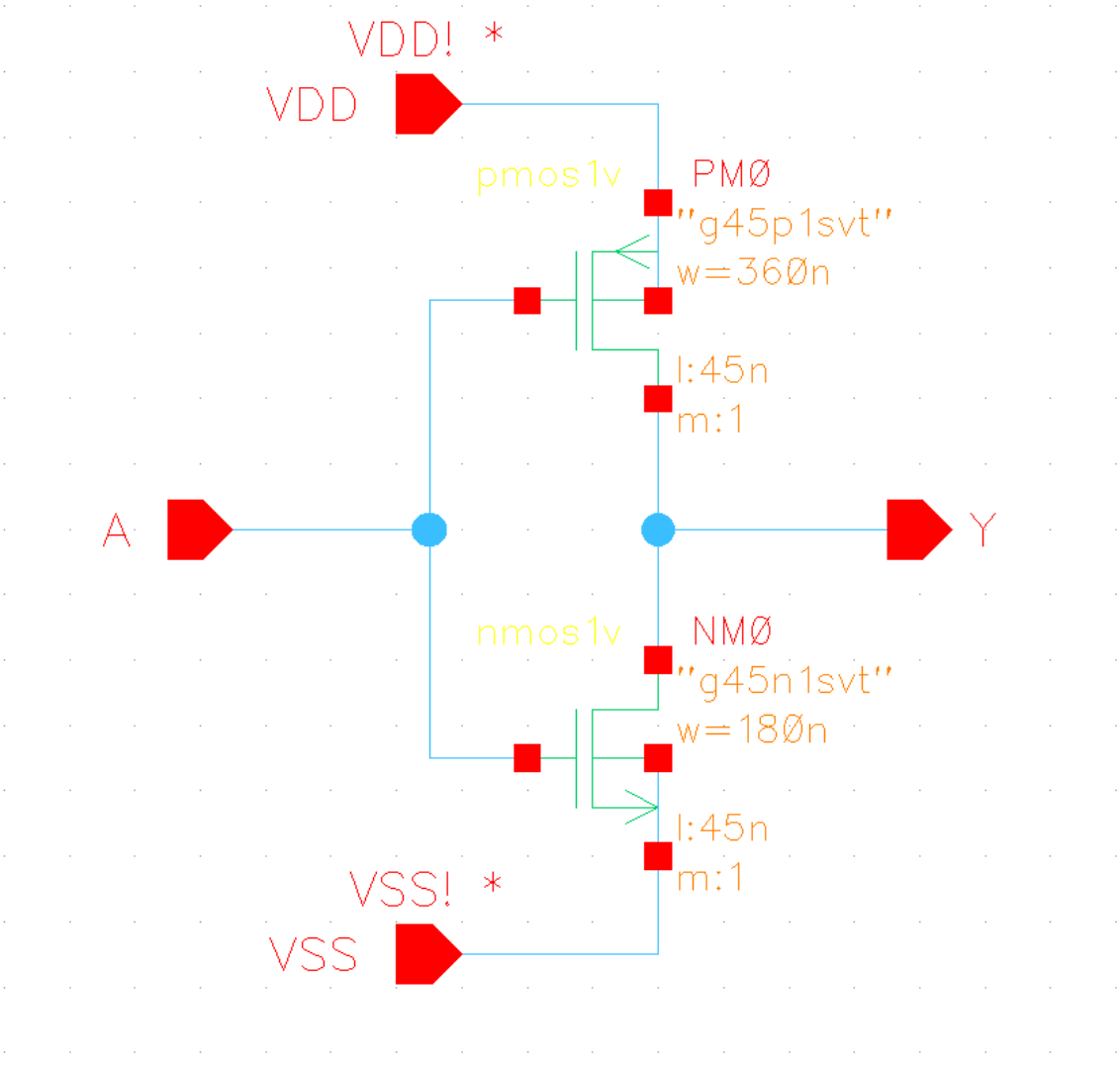

Figure I.1: Schematic of the INV

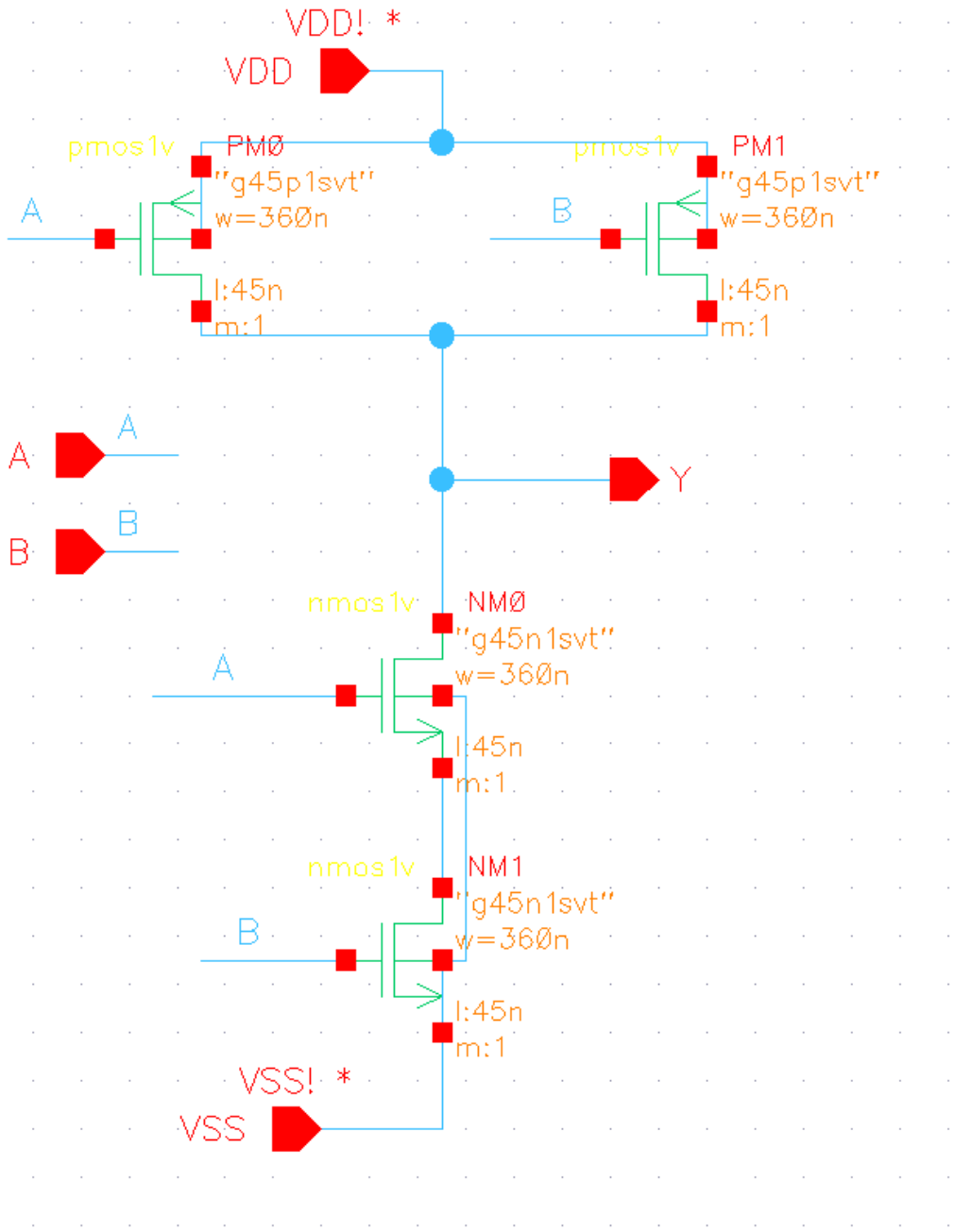

Figure I.2: Schematic of the NAND

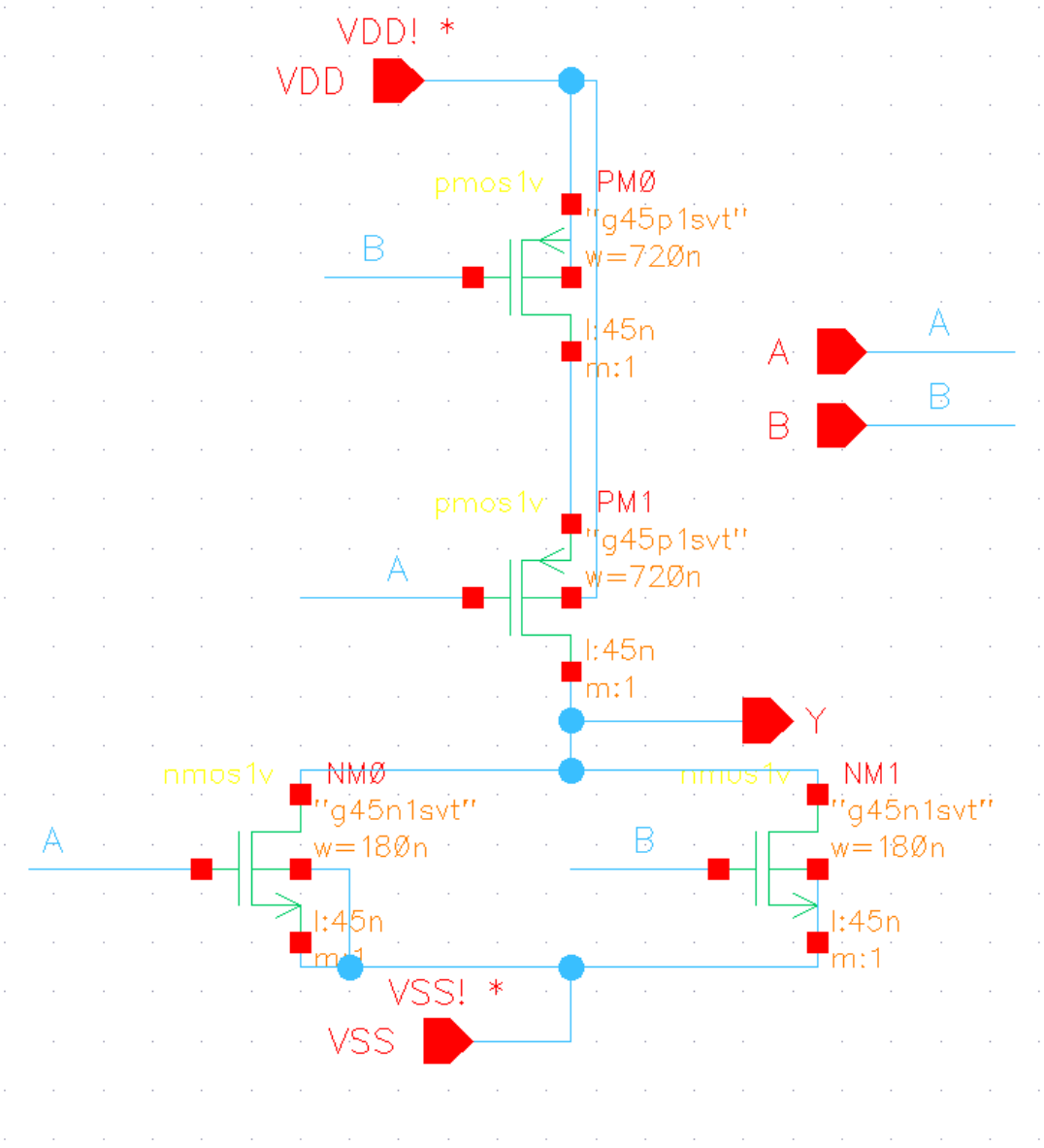

Figure I.3: Schematic of the NOR

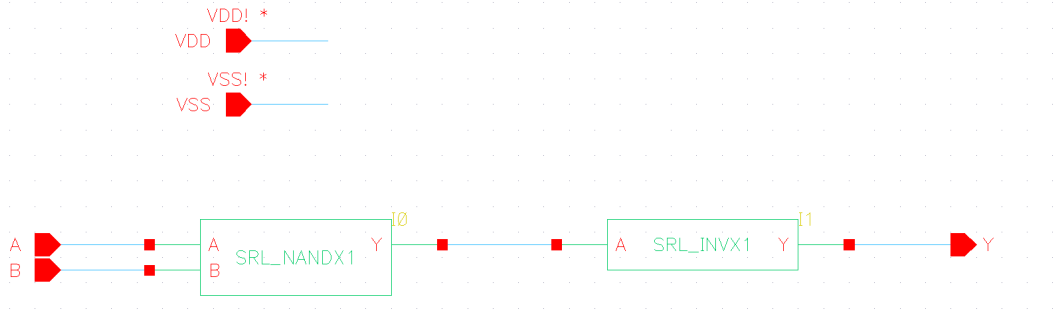

Figure I.4: Schematic of the AND

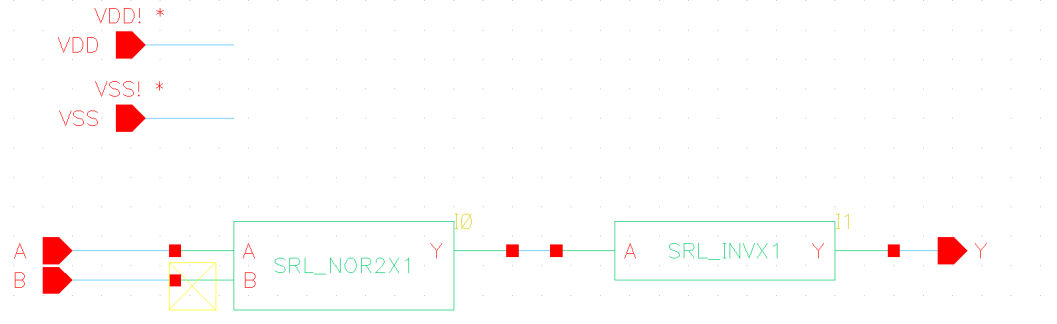

Figure I.5: Schematic of the OR

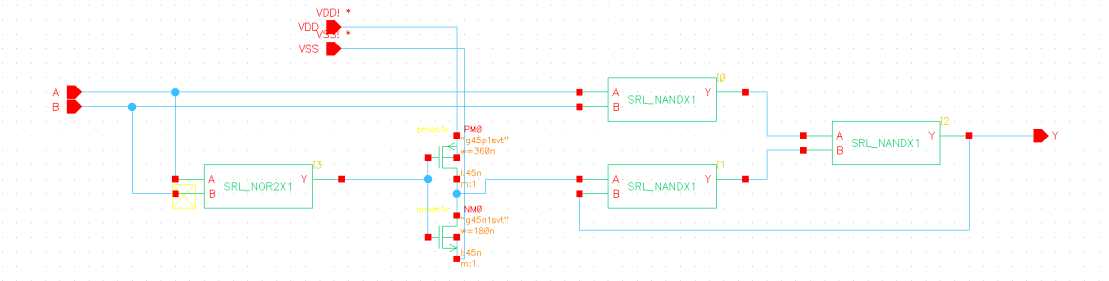

Figure I.6: Schematic of the C-Element

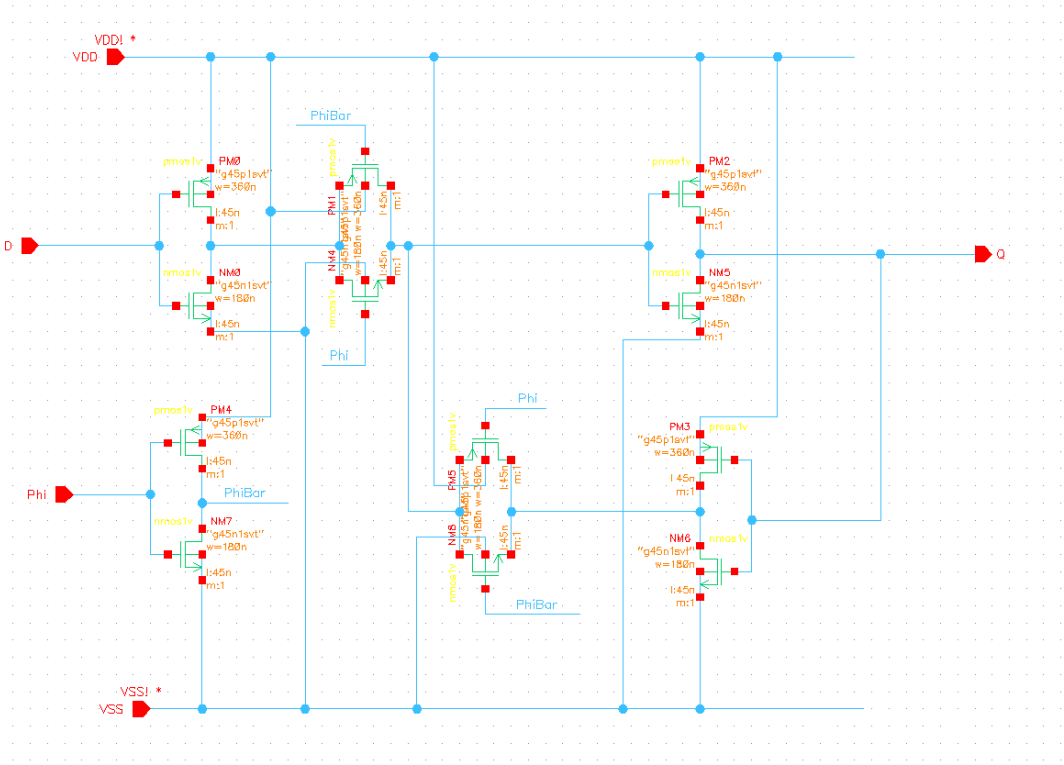

Figure I.7: Schematic of the positive edge triggered LATCH

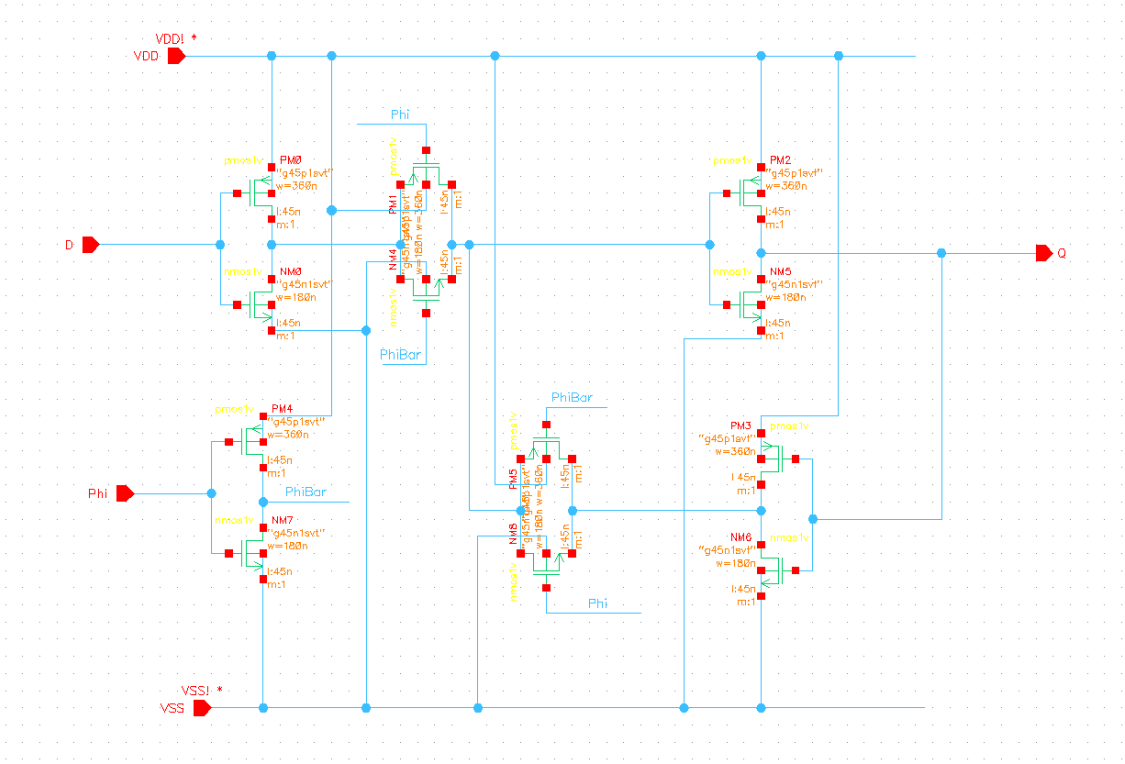

Figure I.8: Schematic of the negative edge triggered LATCH

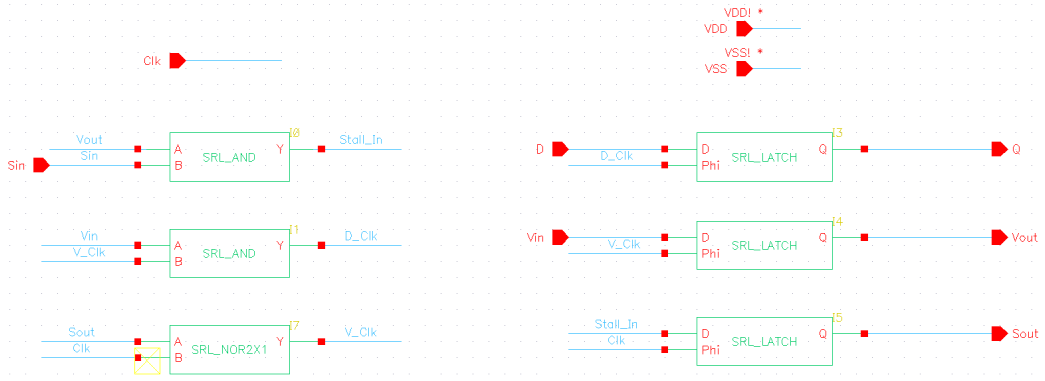

Figure I.9: Schematic of the ISP Stage

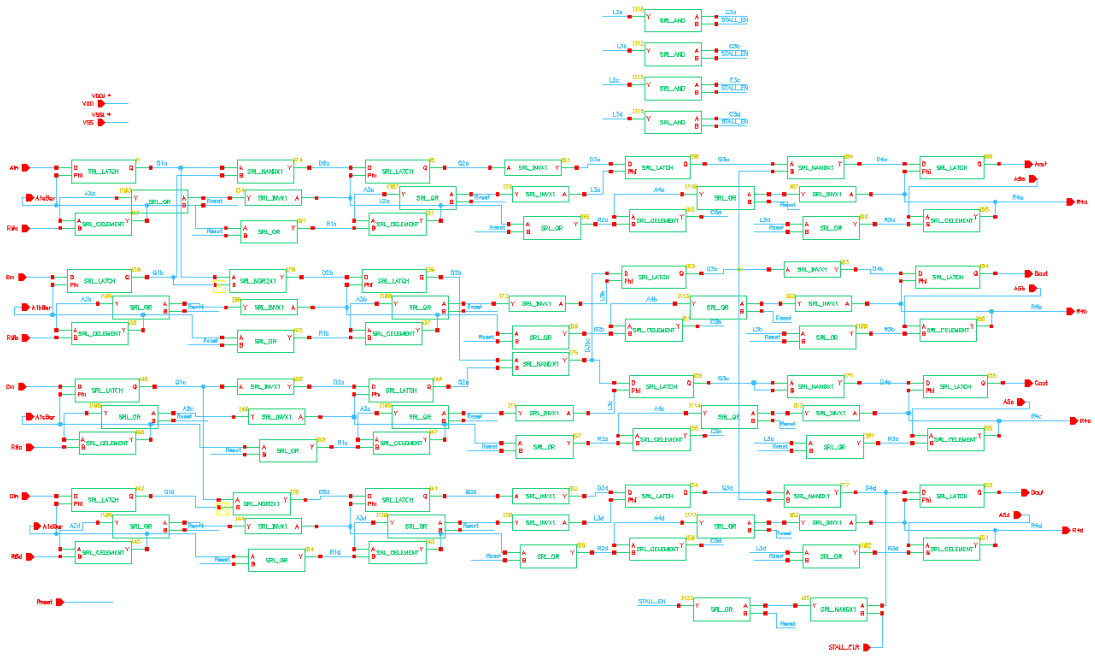

Figure I.10: Schematic of the Asynchronous Pipeline

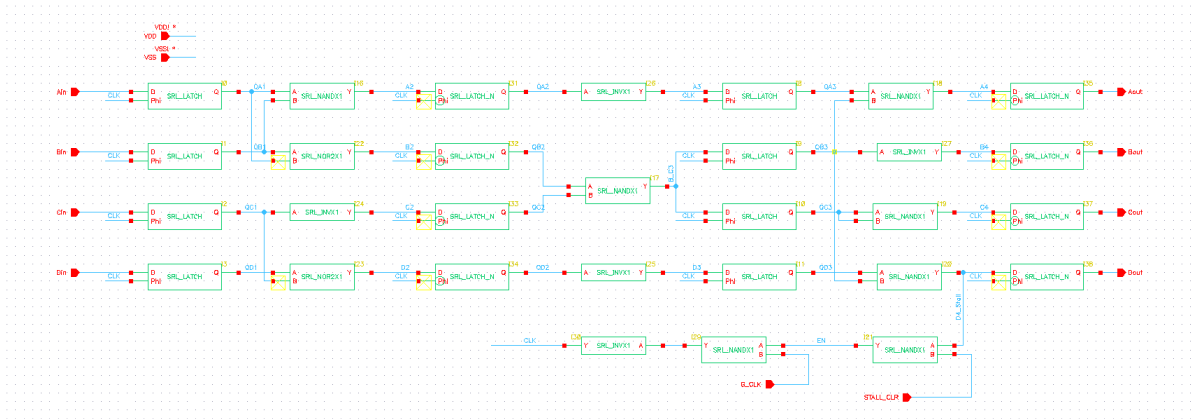

Figure I.11: Schematic of the Synchronous Pipeline

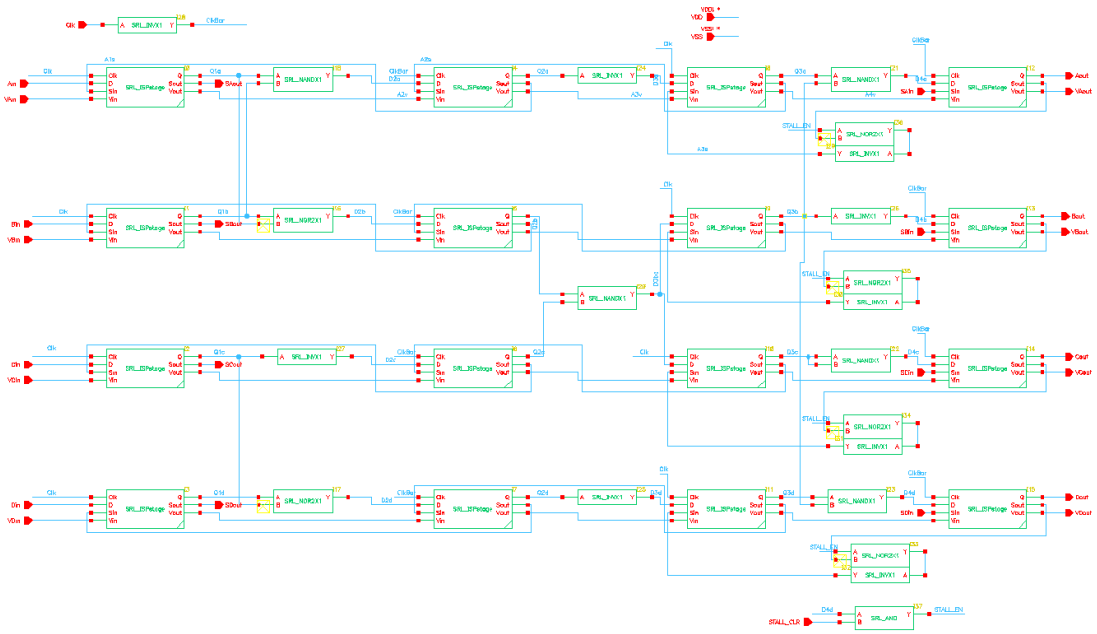

Figure I.12: Schematic of the Interlocked Synchronous Pipeline

Appendix II

Layouts

This appendix contains the layouts for all of the gates and pipelines created for this project.

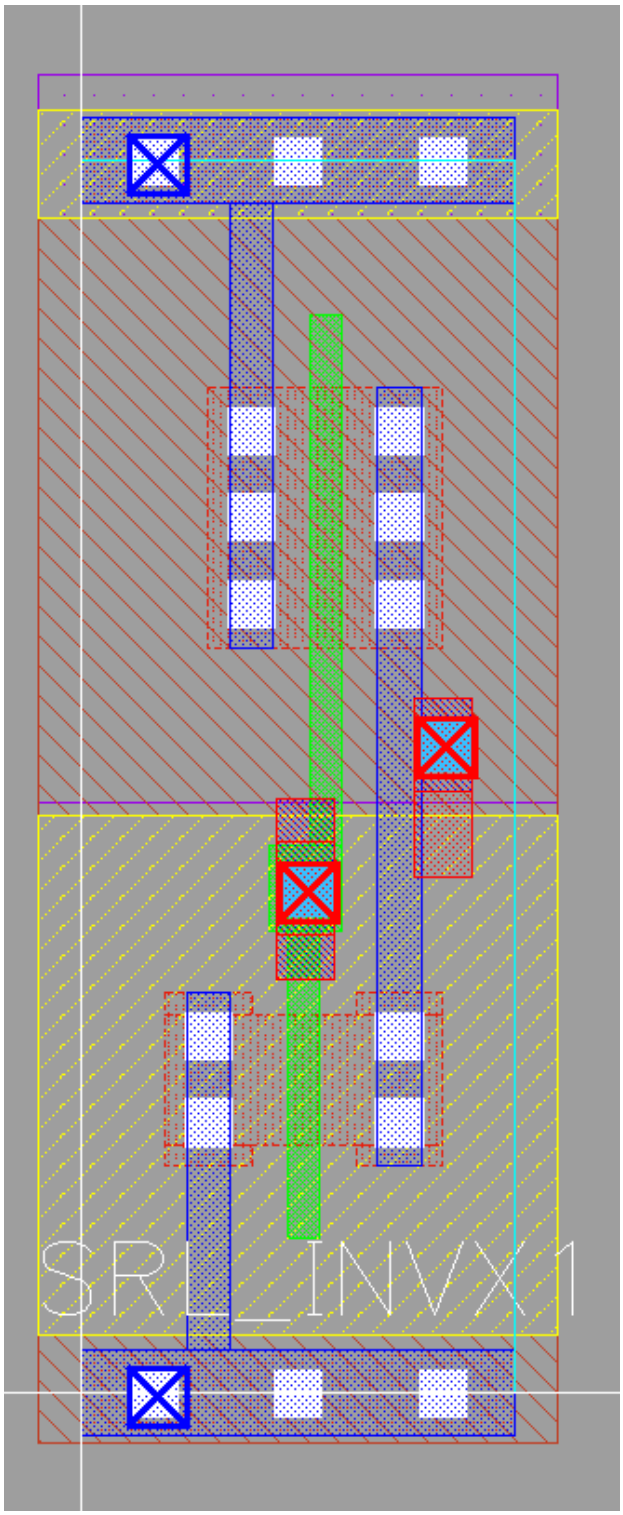

Figure II.1: Layout of the INV

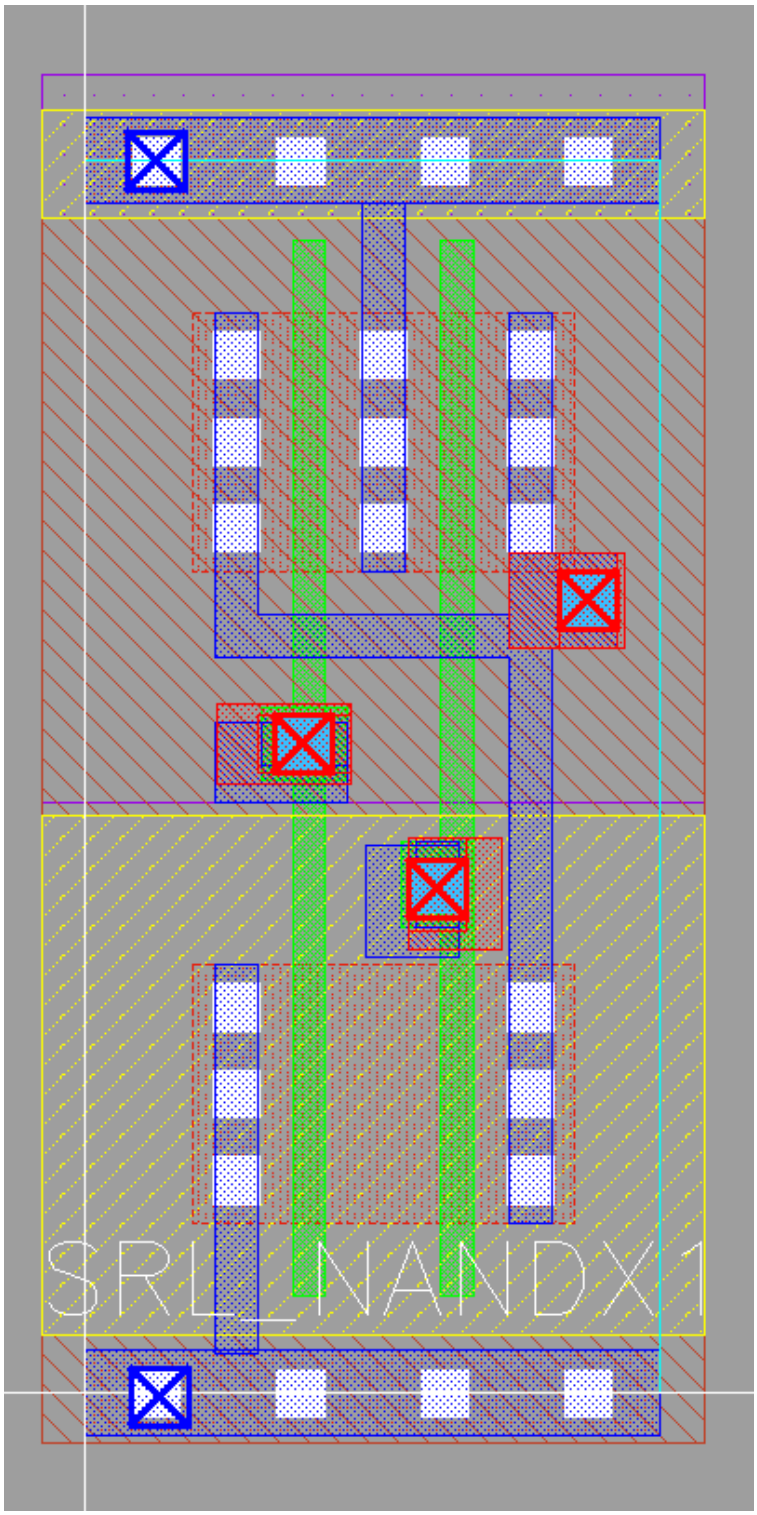

Figure II.2: Layout of the NAND

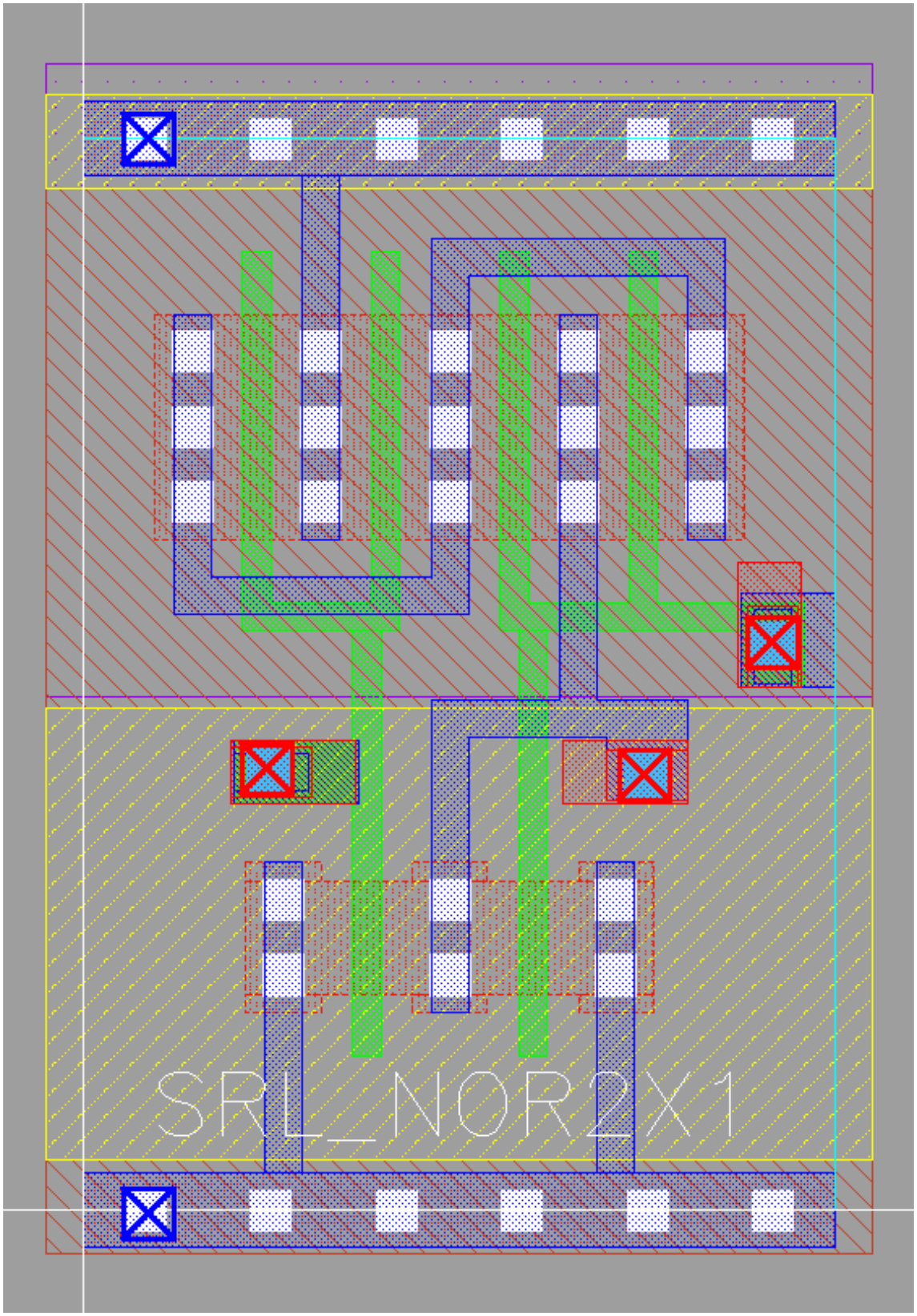

Figure II.3: Layout of the NOR

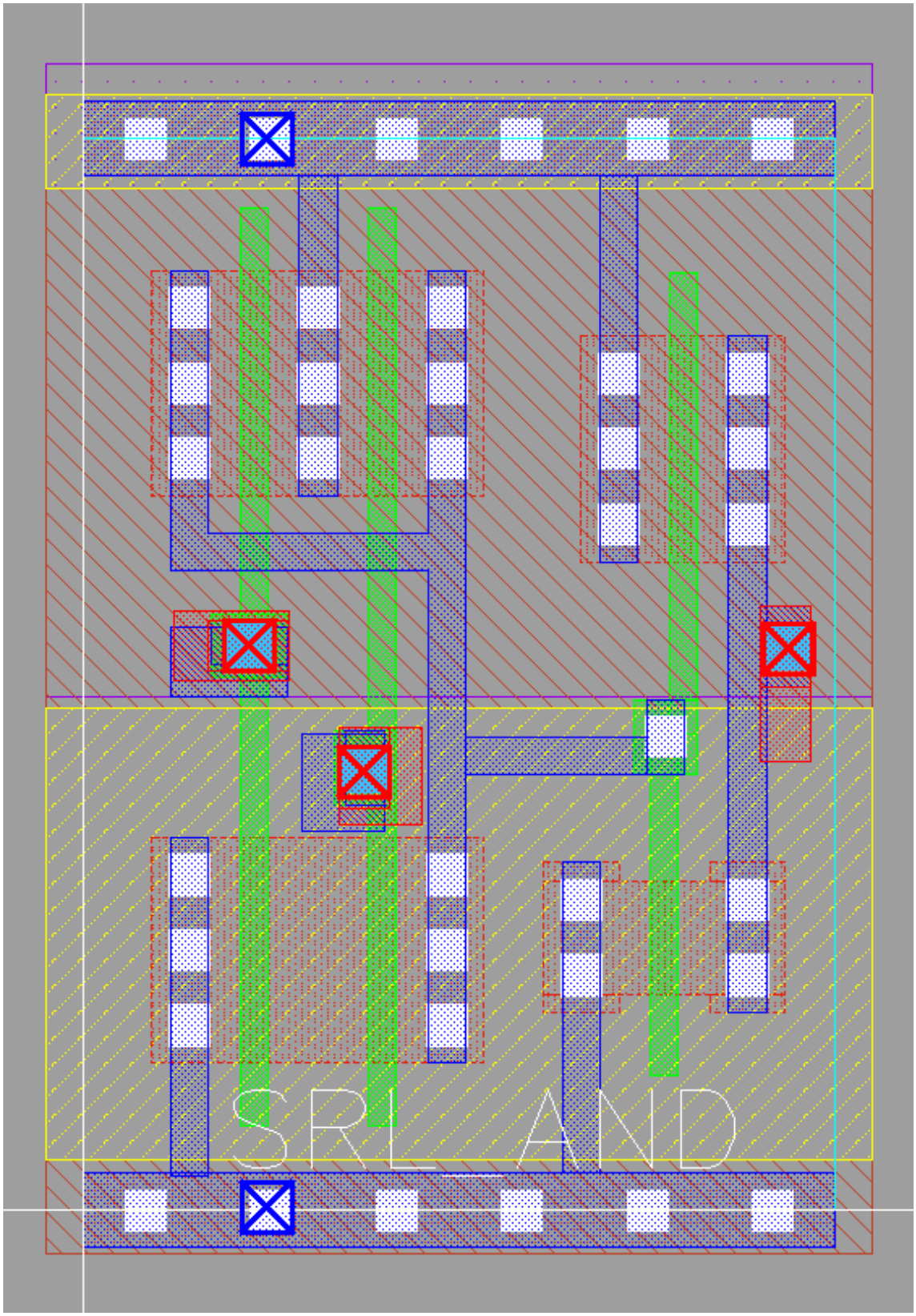

Figure II.4: Layout of the AND

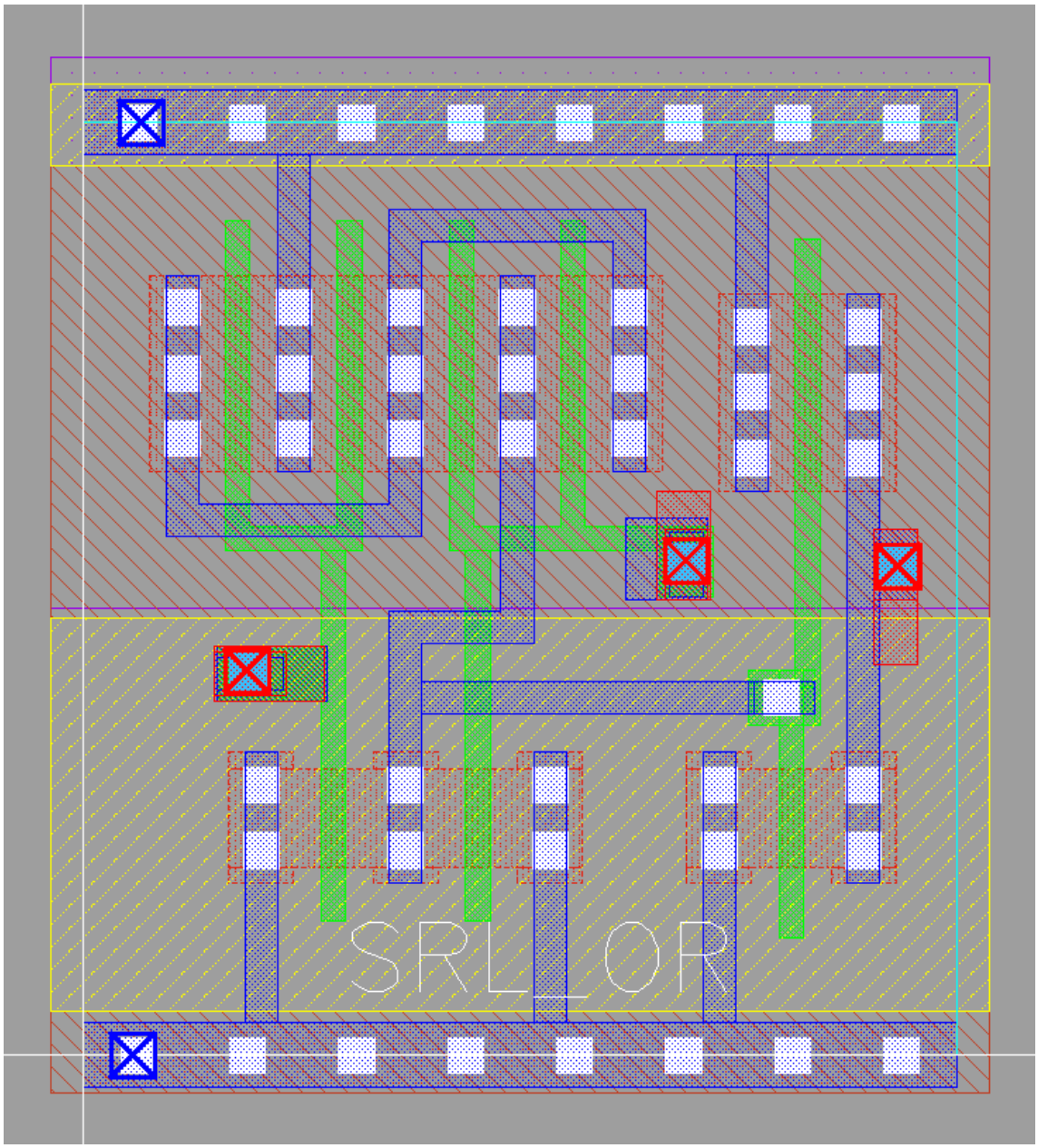

Figure II.5: Layout of the OR

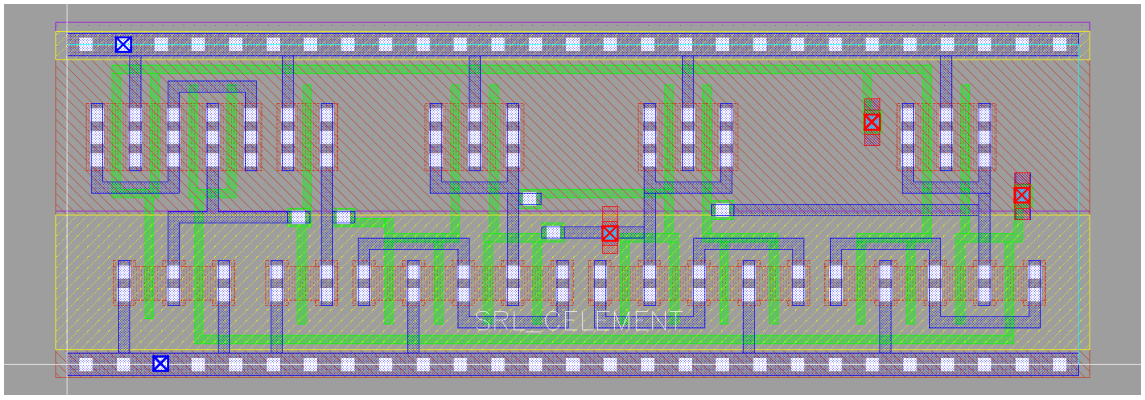

Figure II.6: Layout of the C-Element

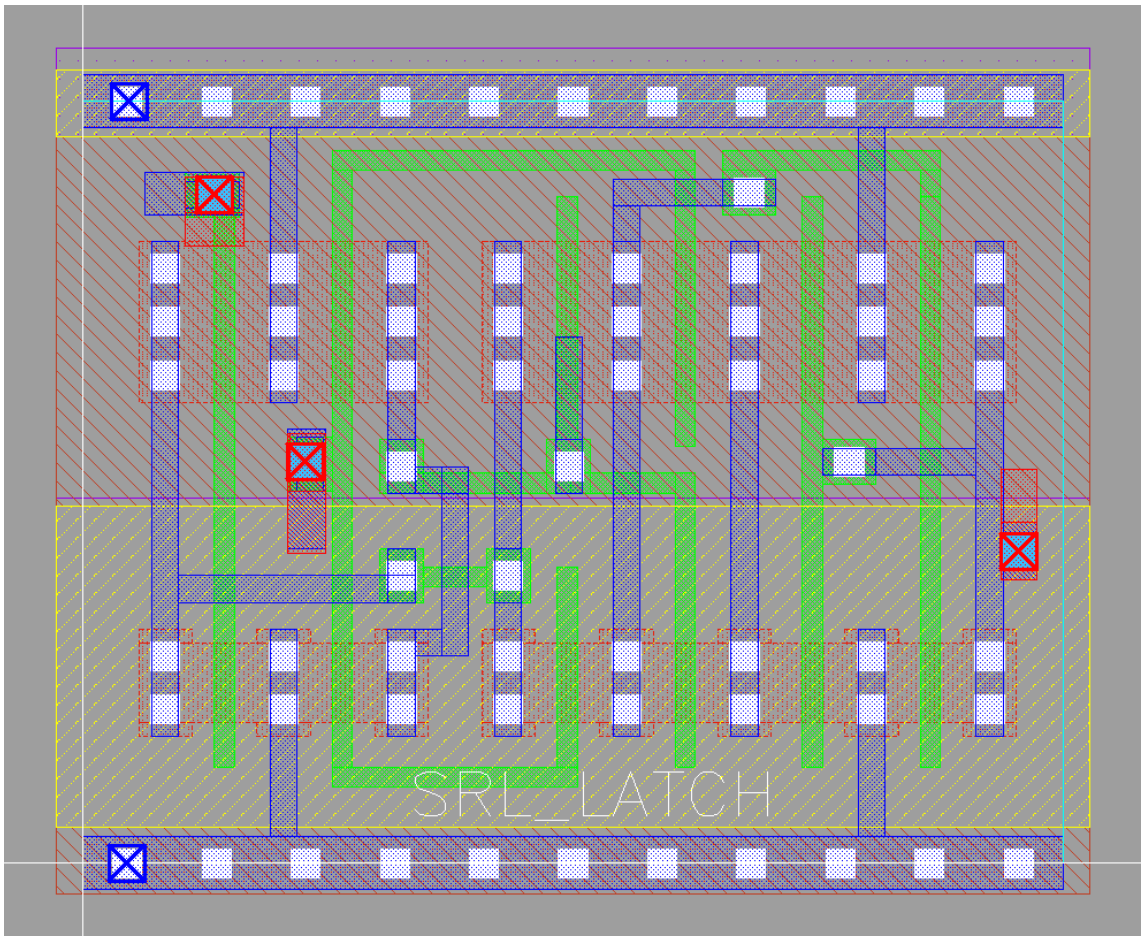

Figure II.7: Layout of the positive edge triggered LATCH

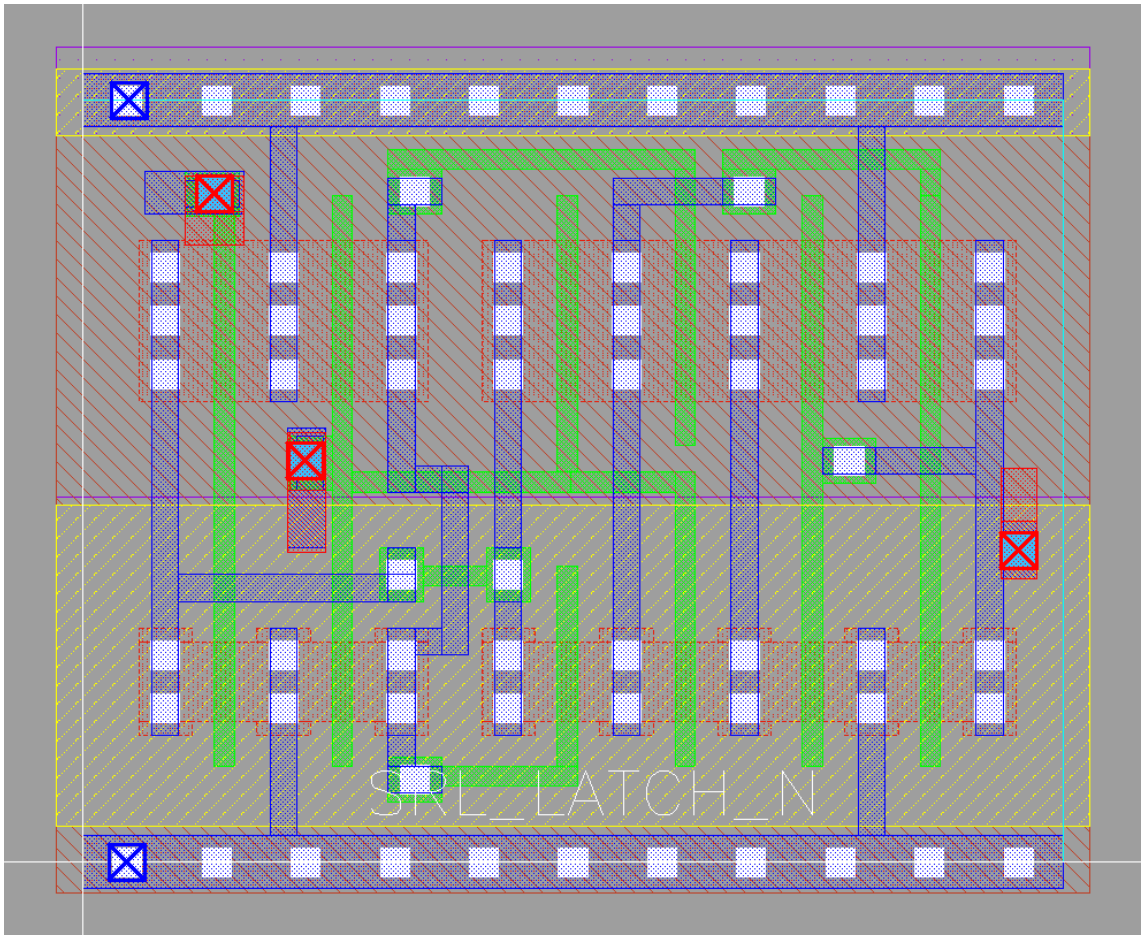

Figure II.8: Layout of the negative edge triggered LATCH

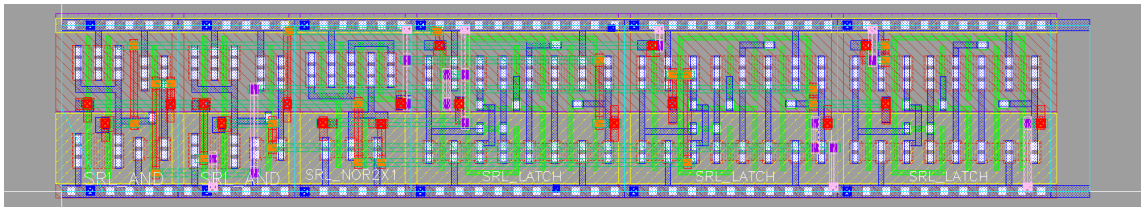

Figure II.9: Layout of the ISP Stage

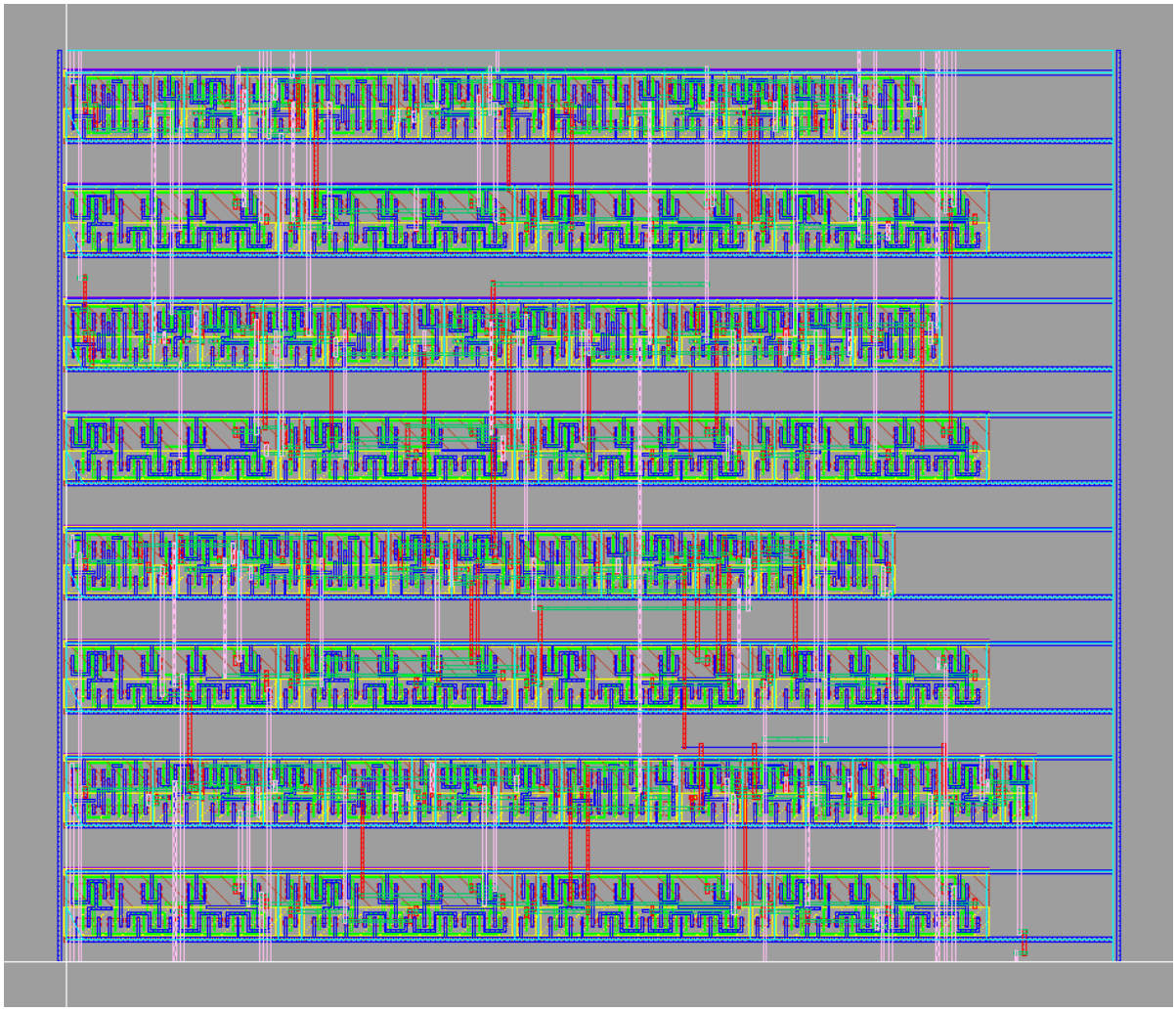

Figure II.10: Layout of the Asynchronous Pipeline

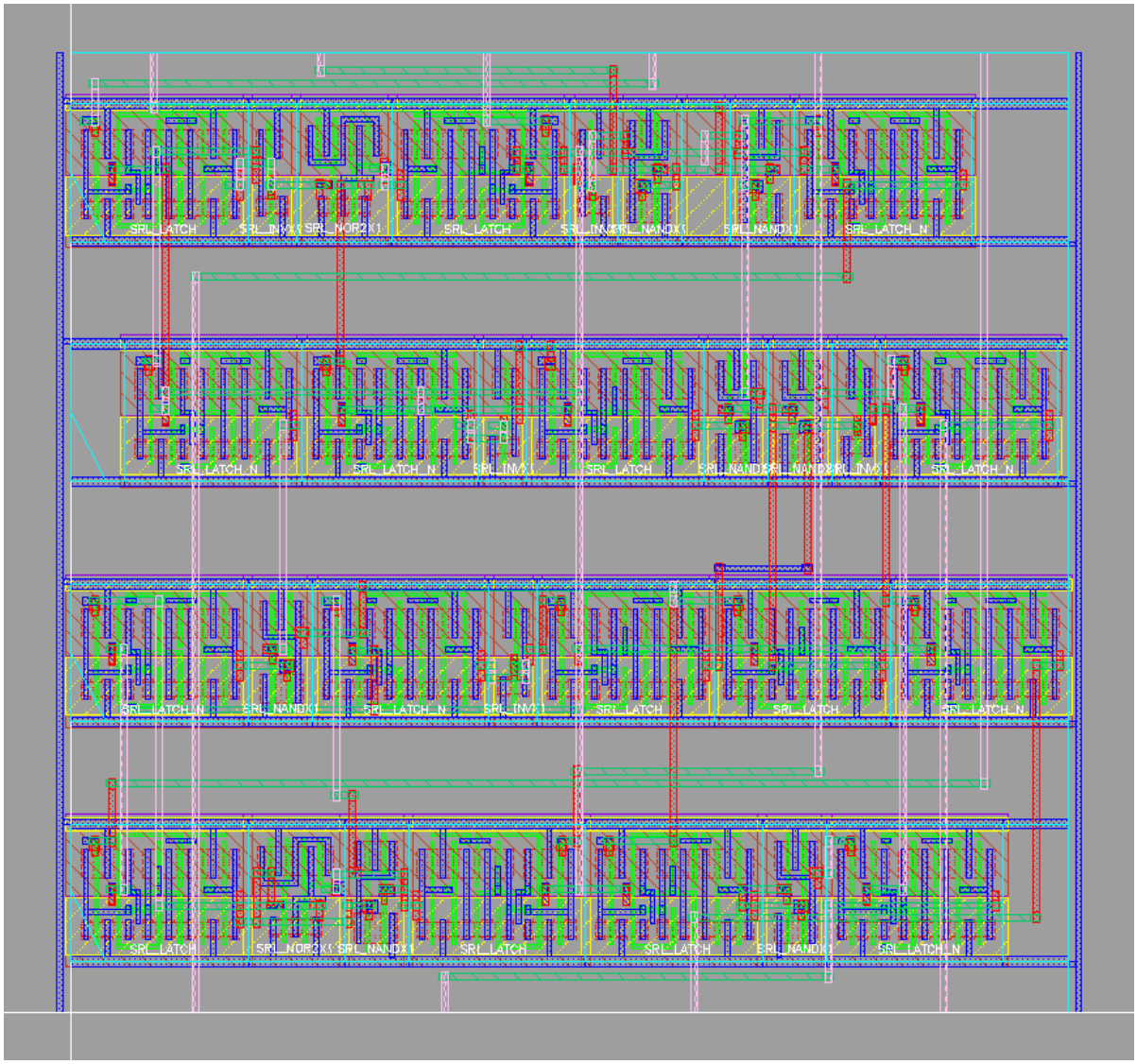

Figure II.11: Layout of the Synchronous Pipeline

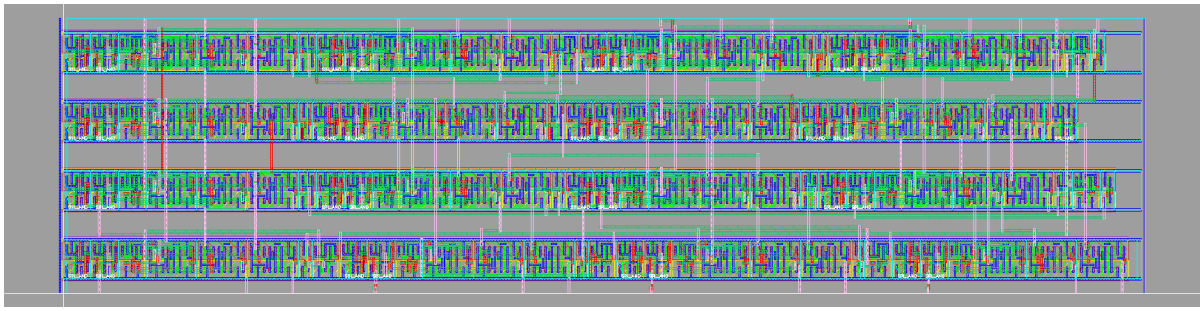

Figure II.12: Layout of the Interlocked Synchronous Pipeline