Audun Asdal

A Low Power Parallel RISC-V Processor in 22nm FDSOI Technology for Medical Ultrasound

Master's thesis in Electronic Systems Design and Innovation
Supervisor: Trond Ytterdal
June 2022

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Audun Asdal

A Low Power Parallel RISC-V Processor in 22nm FDSOI Technology for Medical Ultrasound

Master’s thesis in Electronic Systems Design and Innovation
Supervisor: Trond Ytterdal
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Abstract

Medical ultrasound data is traditionally filtered in the probe using analog filters before sending the less extensive filtered data to an external processing device. By digitizing the signal from the ultrasound transducers, doing digital filtering and processing directly in the probe, the programmer has access to more unfiltered data. The big problem is the tight power budget at 3W set to keep the probe relatively cold, lower than 40°C, while at the same time process the data from 10 000 transducers at a 10MHz sample rate. Picorv32, an area effective RISC-V based processor, has in this project been rebuilt as a multicore processor in order to perform this massive processing problem. The design is synthesized in a 22nm FDSOI (Fully Depleted Silicon On Insulator) technology to extract estimates for power and area.

The design is based on distributed memory, where each core has its own memory block. This works very well for the primary digital filtering of the signals as the data from each probe belongs to a single core. As there is no interconnection between the cores implemented, problems arise when data sharing between the cores is needed. The processor is able to do matrix multiplication at 6.9 CPI (Clocks Per Instruction) per core, with a dynamic power estimate of 12.3µW per MIMD (Multiple Instruction Multiple Data) core and 12.1µW per SIMD (Single Instruction Multiple Data) core at a 10MHz clock frequency. At 500MHz the power estimates are 605µW for the SIMD cores and 613µW for the MIMD cores. The area per core is estimated to 5800µm^2 per MIMD core and 5500µm^2 per SIMD core. As these power estimates are relatively low compared to the 3W power budget even with many cores, this type of system could prove useful in future ultrasound probes.

Sammendrag

I medisinsk ultralyd blir signalene tradisjonelt filtrert relativt kraftig med analoge filtre i proben før de sendes videre til en ekstern prosesseringsenhet. Ved å i stedet digitalisere signalene og deretter gjøre den digitale prosesseringen internt i selve proben vil programmereren ha tilgang til mye mer ufiltrert data. Det store problemet med å skulle gjøre hele prosesseringen i proben er det lave effektbudsjettet på knappe 3W for å holde proben under 40°C og samtidig prosessere data fra 10 000 prober med en punktprøvingsfrekvens på 10MHz. PicoRV32, en arealeffektiv RISC-V-basert prosessor er i dette prosjektet ombygd til å støtte flere kjerner for å løse dette problemet. Designet er syntesert i en 22nm FDSOI (Fully Depleted Silicon On Insulator) teknologi for å estimere effekt og størrelse.

Designet er satt opp med et distribuert minne hvor hver kjerne har en egen minneblokk. Denne løsningen fungerer bra for å filtrere signalene fra transducerne siden hver transducer hører til under en enkelt kjerne. Siden det ikke er noen kommunikasjon mellom kjernene, blir det problemer når det trengs data fra flere kjerner. Prosessoren kjører matrisemultiplikasjon på 6.9 CPI (Clocks Per Instruction) per kjerne med en dynamisk effekt på 12.3µW per MIMD (Multiple Instruction Multiple Data) kjerne og 12.1µW per SIMD (Single Instruction Multiple Data) kjerne ved en klokkefrekvens på 10MHz. Ved 500MHz klokkefrekvens bruker de henholdsvis 613 og 605µW per kjerne. Areale per kjerne er estimert til 5800µm^2 per MIMD kjerne og 5500µm^2 per SIMD kjerne. Siden disse effektestimatene er relativt lave i forhold til budsjettet på 3W selv med mange kjerner vil denne type system være interessant i fremtidige ultralydprober.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Ultrasound Processing

Ultrasound is an immensely useful tool in order to observe underneath the skin of a patient without harm. The basic idea is quite simple. One or more transducers sends an ultrasound signal into the tissue, that is, a sound wave with a higher frequency than what a human ear can hear. A part of this wave will be reflected by different parts of the tissue, back to the transducer(s). A simple diagram of an ultrasound transducer array with eight transducers in a row is shown in figure 1.1. Reflections happen where the density of the tissue changes [1].

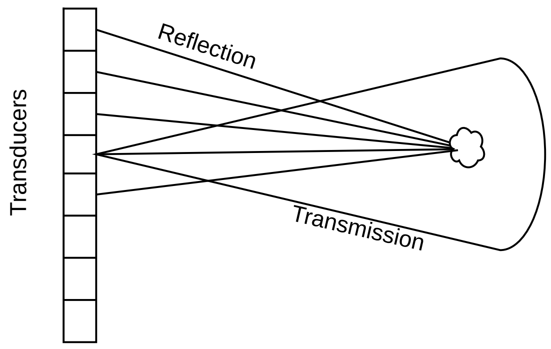

Figure 1.1: A simplified overview of an ultrasound transducer array.

Here we see how a signal is transmitted by transducer number four from the top, and how the signal is reflected as it hits an entity. By processing the reflected signal hitting each transducer, one may find the position of this entity, as well as the speed by finding the doppler shift of the transmitted signal [2]. This is the basic idea of ultrasound imaging, but as the tissue is vastly more complex than a single entity in vacuum, and the transducer array could consist of thousands of transducers in several dimensions, ultrasound imagery is not that simple.

Traditionally, ultrasound processing needs a lot of computing power from an external rack. Given the small size of an ultrasound probe, this massive computation has been mostly done in a later stage, outside the probe. In order to be able to transfer data from the probe to this processor, a fat cable between the probe and the processing unit is needed. But the amount of data produced by the probe is still too large, and some analog preprocessing is done in the probe to drastically reduce the amount of data transferred. The problem with this technique is that the original data is altered. Depending on the loss in the analog preprocessing, the processor outside the probe has limited data to create the image [3].

In figure 1.2 we see how ultrasound systems are traditionally built. Some analog preprocessing is done in the probe, while a lot of data is sent across a large cable to be processed in a dedicated machine. This powerful processor can be reprogrammed and changed in order to create different images, which makes this approach quite flexible [3].

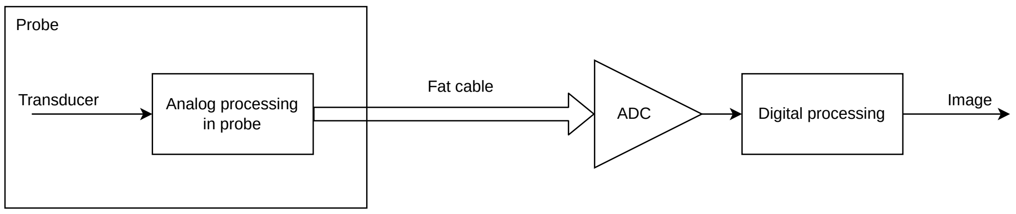

Figure 1.2: Block overview of a traditional ultrasound processing. Analog preprocessing to reduce the data rate across the cable to the main processing.

As technology scales, more and more processing can fit inside the probe, and at some point most of the processing could be done internally in the probe. By doing the processing in the probe itself, the loss in the preprocessor is not needed, and all the original data is present for the internal processor. This solution is shown in figure 1.3.

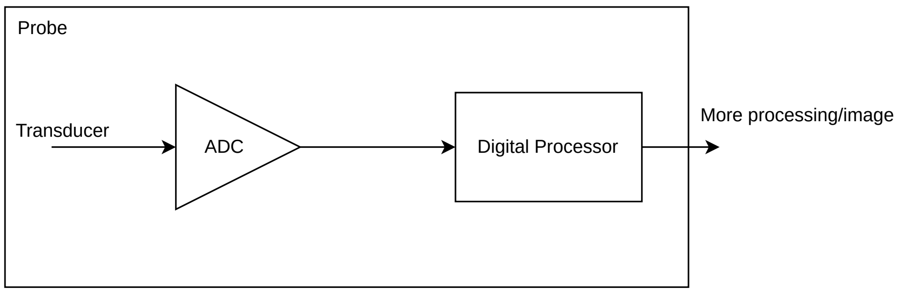

Figure 1.3: By moving the processing into the probe, all original data is available to the processor.

A probe with all processing internally is most wanted, creating the final image directly. This way the digital signal processing programmers have all the original data at their fingertips. There is also no need for sending the data large distances. By having all the data available for processing in the probe, the imaging could have better quality, while also reducing energy and cost. The problems, however, are the strict energy and area limitations in the probe. A probe used for medical ultrasound imaging has strict temperature demands, so as not to inflict burns on the patient. This in turn implies a strict power limitation [4].

1.2 Existing Technology

1.2.1 Handheld Probes

A number of handheld devices are already on the market and are possible to buy for the general consumer. Still, there are limitations to these probes. For one, the image quality is inferior to the larger systems. The processing power in the probe itself just isn't good enough for top-quality imaging. The biggest problem, however, seems to be the temperature. These smaller handheld probes do a lot of processing in the probe, and thus dissipate heat. This results in the probe overheating and limits the time it can be used. The probes turn off when the temperature is too high, and cannot be used again until they have cooled down [5].

1.2.2 Traditional Ultrasound Systems

The alternative is the more traditional systems. These have all the processing power and cooling needed, as well as the flexibility of reprogramming the system. The problem with these systems is, as mentioned, the bottleneck that is the cable. A lot of data is lost in order to be able to send it to the main processing unit across this cable. These systems are also large and expensive, too expensive for all doctors to have access to such a system.

1.2.3 RISC-V

One way of creating a flexible processing element for the probe is to use a processor. The relatively new open source instruction set RISC-V [6] could be ideal for this. An advantage of the instruction set being open source is the many open source implementations available. This makes it possible to choose a low-power implementation as a base for the design.

One implementation of a massively parallel risc-v based processor is the GRVI (Gray Risc-V Implementation)[7]. This implementation is used to accelerate the connection between the main processor and accelerators connected to it. It is implemented on a Xilinx FPGA, with 400 cores. This implementation uses as little as 13W running full-speed. In order to communicate between the cores, a NoC (Network on Chip) is created. A simple two-dimensional router is used, which will not accept data until the router has the capacity. This solution is shown to work adequately as long as the traffic over the router remains quite low compared to the theoretically possible throughput.

1.2.4 Custom Hardware Implementation

Another possibility is to create a custom hardware implementation. This solution would be tailor-made to the probe, and would not be possible to change much after production. However, the solution would probably be more energy efficient, as no unneeded circuits would lie dormant. The choice between custom hardware or a more general processor is a tradeoff between power and flexibility with the possibility to change and update the probe continuously.

1.3 The Goal

By minimizing the power consumption of the image processing, it could be possible to bring the image quality of the high-end large ultrasound systems into the small probe. The image quality could even be improved, as no preprocessing is needed for sending across a cable. This means all the data is available to the programmer, uncompressed. With more data, the programmer may pick and choose more freely, and this may result in even better ultrasound images.

As it would be a huge advantage to be able to change the processing algorithm of the probe continuously, it seems the best solution is processor-based rather than custom hardware. This solution will probably consume more power, but if the power is low enough, the result will be a way more versatile probe. A probe that can be programmed with specific algorithms to fit the task as well as possible.

1.4 Overview of the Report

The rest of the report starts with a theory chapter, explaining the needed theory. We will here take a look at ultrasound technology before moving over to digital circuits at a low level, with a specific interest in power consumption. Further, we will take a look into parallel processing, and how to modify a RISC-V core to enable it.

The next chapter concerns the multicore RISC-V implementation, and how the single core processor has become a multicore one. This chapter also shows how the processor is tested and explains some limitations of the project. Then comes the results and discussion, where results will be discussed as it is presented. Here as well, with a strong emphasis on power and area. Then a more general discussion part, before moving over to a conclusion and future work for the project.

Chapter 2

Theory

2.1 Ultrasound Beamforming

As seen in figure 1.1 in the introduction, the reflected sound wave may be received by several of the transducers in an array. But the distance from the entity to each transducer is not the same when the transducers are located in a plane. This is where beamforming comes into play. A simple beamforming setup is shown in figure 2.1 under.

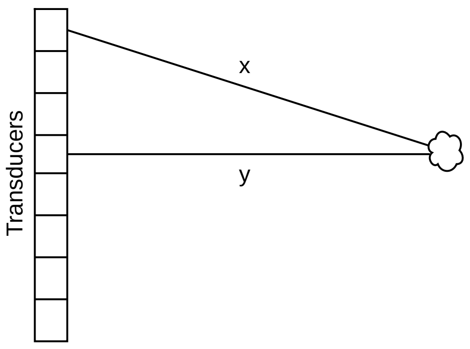

Figure 2.1: A simple beamforming array with different distances x and y to two transducers.

From this figure, which shows a linear one-dimensional array with eight transducers, we see how the distance x to the topmost transducer is longer than the distance y. In order to be able to sum these signals in phase, the signal traveling along y will have to be delayed before summing the two signals. However, as the distance between the transducer elements is much smaller than the distance from the probe to the point of interest, the reflected signal may be simplified to be a plane wave reaching the probe, as shown in figure 2.2 [8].

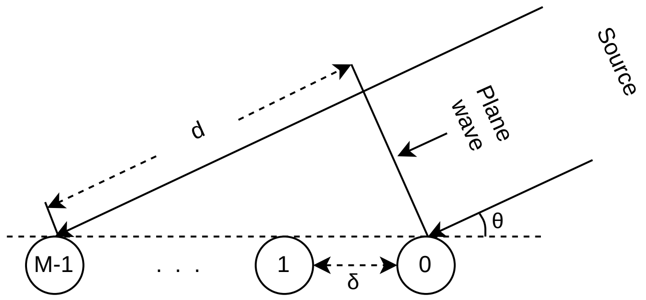

Figure 2.2: Plane waves reaching a transducer array at an angle θ.

The distance d is shown in (2.1) [8].

$$d = (M-1)\delta \cdot cos(\theta) \tag{2.1}$$

As seen in figure 2.2, δ is the distance between the equally distanced M transducers, and θ is the incident angle of the plane wave. To find the time delay for the plane wave to travel the distance d, the only additional information needed is the speed of sound c (2.2).

$$t = \frac{(M-1)\delta \cdot cos(\theta)}{c} \tag{2.2}$$

With a known sampling frequency F_s of the array, one may now add a delayed sample from transducer 0 with a sample from transducer M-1 in order to perform beamforming at a specific angle θ. The connection between time delay t and the number of samples n is given by (2.3) [9]. T is alternatively the sample period. By using several transducers, the accuracy improves further.

$$t = nT = \frac{n}{F_s} \tag{2.3}$$

As previously mentioned, the idea behind beamforming is quite simple, yet when using an array with several dimensions, and thousands of transducers, the calculations become quite heavy. What is needed to perform beamforming is the ability to sum weighted samples from different transducers at different delays.

Further on we will look at what is needed to implement such a system capable of individually setting the wight and delay of each sample before summing them.

2.2 Low Level

2.2.1 Power in Digital Circuits

The power dissipated in a digital circuit is divided into two main groups as shown in (2.4) [10]. These two parts are the dynamic and the static power respectively. The dynamic power is the power dissipated as a result of switching in the circuit and will contribute most in periods when the circuit has a high toggle rate.

Static power is the power dissipated simply from having the circuit turned on, that is, from sources such as resistive coupling between the power source and ground, and quantum mechanical tunneling. In nanometer technologies, leakage power contributes to about a third of the total power dissipation in a typical design [10].

$$P_{total} = P_{dyn} + P_{static} \tag{2.4}$$

The dynamic power can be further divided as in (2.5).

$$P_{dyn} = \alpha f C V_{DD}^2 \tag{2.5}$$

Here, α is the activity factor of the circuit, how much of the circuit is toggling at a given time. C is the total output capacitance of the circuit, and f is the clock frequency of the circuit. V_{DD} is the supply voltage [10].

From equation 2.5 we see how lowering any of the factors will result in lower dynamic power. In addition, as the frequency is proportional to the voltage needed, a change of the clock frequency will also alter the needed voltage. And vice versa, a lowering of the supply voltage limits the highest possible clock frequency [10, 11].

The static power dissipation has the form we see in (2.6).

$$P_{static} = (I_{sub} + I_{gate} + I_{junct} + I_{contention}) \cdot V_{DD} \tag{2.6}$$

We see how there are several components contributing to the static power dissipation, depending on where the current flow goes. The one common denominator is the supply voltage V_{DD}. The currents are shown in figure 2.3. In CMOS (Complementary Metal Oxide Semiconductor) technology there is no contention current [10].

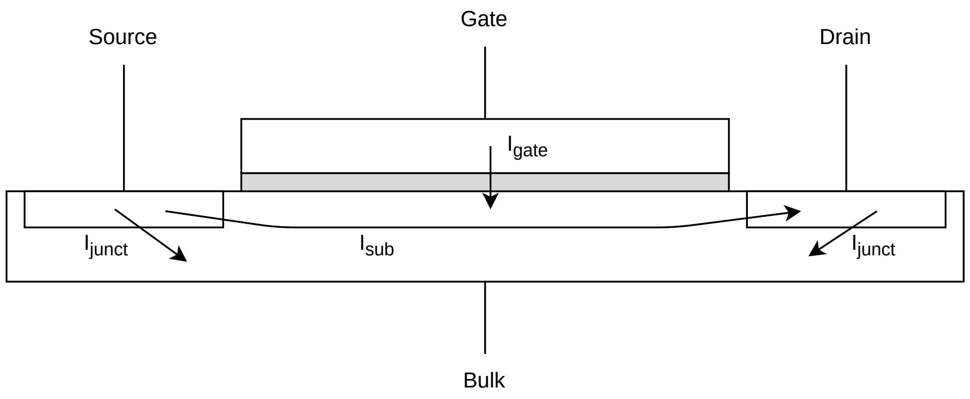

Figure 2.3: The currents contributing to the static power dissipation in CMOS.

The subthreshold leakage I_{sub} flows across the transistor from source to drain when the transistor is supposed to be off. Some current will still flow, and this current increases drastically with the temperature. When the transistor is in the subthreshold region, the V_{gs} voltage swing is not particularly high, and the transistor will have a higher subthreshold leakage current as it cannot turn the transistor completely off. A higher source voltage or lower bulk voltage may reduce this leakage. FDSOI technology has a relatively low subthreshold leakage due to its sharp subthreshold rolloff [10].

Gate leakage is the current leakage from gate to bulk. When the dielectric insulation between the gate and bulk shrinks with technology, the dielectric becomes extremely thin. So thin in fact, that charge carriers tunnel through it, resulting in the I_{gate} leakage current. Naturally, this current depends on the area and thickness of the dielectric, as well as the voltage across the gate. Finally, the I_{junct} current is the current from the source or drain to the base. As these are reverse biased diode connections, this current is almost negligible in FDSOI technology [10].

In both static and dynamic power dissipation the supply voltage plays a big role in the power consumption of the circuit. As the clock frequency is closely connected with the supply voltage, the clock frequency dictates how low the supply voltage can be set. Additionally, the area of the design plays a big role, especially for the static power consumption. A larger area often means more power. We will now look into some techniques for lowering the power consumption of a digital circuit.

2.2.2 Clock Gating

One simple way of limiting the dynamic power dissipation as seen in (2.5) is to lower the activity factor α. This can be done by inserting clock gates into the circuit. These gates, as shown in figure 2.4, stop the clock from reaching parts of the circuit using a control signal [12].

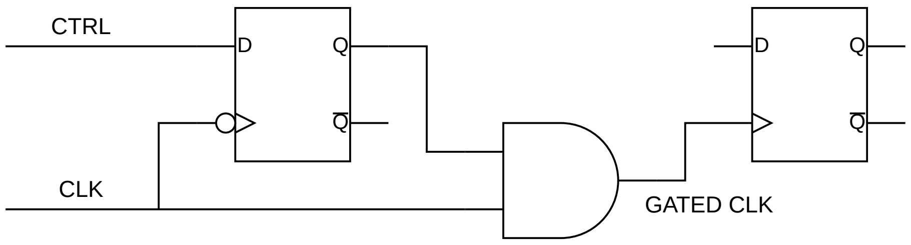

Figure 2.4: A latched AND clock gating block.

The latch ensures no glitches can pass the gate, causing errors and excessive switching in the gated signal [12]. This solution of gating the clock to a part of the circuit works on the α factor from (2.5). When the clock gate stops the clock from advancing, this factor becomes zero in this gated sub-circuit and no dynamic power is dissipated.

Another benefit of clock gating is in the clock tree itself. The clock tree has a huge fanout and several buffers to feed this large wire. The clock is ultimately a wire. And as it is fanned out, it becomes huge and has a large capacitance. By gating the clock, this fanout is limited in certain areas, lowering the total capacitance of the active clock net.

2.2.3 Power Gating

As seen in (2.6), the static power consumption is closely connected with the supply voltage V_{DD}. Hence, by lowering the voltage as much as possible, the static power will be minimized. If, however, a part of the design is inactive for a longer period of time, the power supply to this whole part may be turned off. This is called power gating and is shown in figure 2.5 [10].

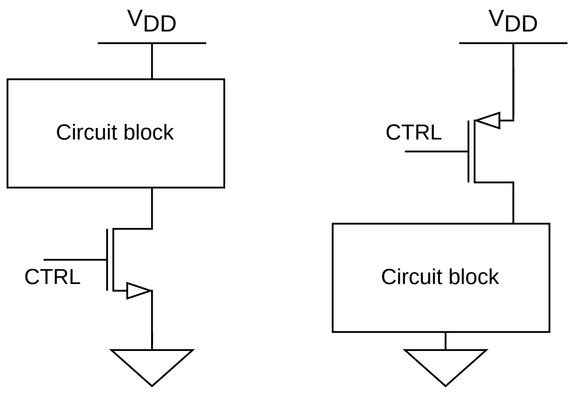

Figure 2.5: Power gating with a footer switch and a header switch respectively.

As seen in the figure, one may gate either the ground connection or the power supply. Several power gating transistors may be used in parallel in order to reduce the resistance and draw more current to a larger circuit block. The size and number of power gating transistors is a tradeoff between delay in the circuit block and the leakage power when in sleep mode when the power is gated. One problem with power gating is that the gated registers will lose their power and therefore lose their stored information. This may be solved using state retention registers that can hold their value for an extended period of time, even without a power supply connected. Alternatively one may only reduce the supply voltage and not turn it completely off. Then the registers could hold their values even in this half sleep mode. The start-up time from sleep to active would then be shortened as well, as the circuit is already half powered. The price of only lowering the voltage is of course more leakage current than a full power off [10].

2.2.4 Voltage Threshold

The voltage threshold in a transistor used in digital design determines the needed voltage on the gate for the transistor to switch state [10]. A lower voltage threshold means the transistor turns on faster when running at the same supply voltage. This makes the lower threshold voltage transistors better when speed is needed, as they can switch faster. This is illustrated in figure 2.6. Here we see how the transition curve V_i reaches the low voltage threshold Vt_1 much faster than the high voltage threshold Vt_2.

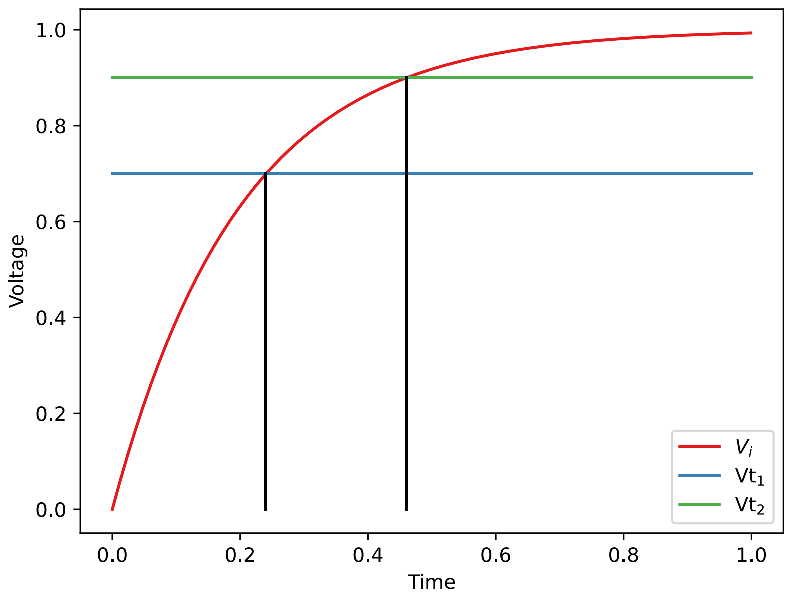

Figure 2.6: Input voltage V_i of a transistor with two possible voltage thresholds Vt_1 and Vt_2.

This comes at a price, though. A low voltage threshold means the transistor leaks more current when it's supposed to be turned off, compared with a high voltage threshold transistor [10].

Another advantage of low-threshold transistors is the possibility for a lower supply voltage. Instead of speeding up the circuit, one could instead lower the supply voltage of the design. As seen in (2.5) and (2.6), lowering the supply voltage could drastically reduce both dynamic and static power consumption.

2.2.5 22nm FDSOI

FDSOI (Fully Depleted Silicon On Insulator) is a type of CMOS technology. One big difference between the FDSOI technology compared to standard CMOS, is the possibility to control the bulk voltage. By controlling the bulk, the voltage threshold of the transistor is altered, making it possible to reduce the voltage threshold when speed is needed, and use a higher voltage threshold to reduce leakage when the circuit is running slowly or in sleep. For standard CMOS the voltage threshold has to be decided when creating the circuit, while for FDSOI it may be done dynamically in this way [13].

2.3 High Level

2.3.1 Multiprocessing

When processing data that is not highly dependent on intermediate answers from other data, we may choose to parallelize the processing. From (2.5) about dynamic power dissipation in digital circuits we see how the power is proportional to the clock frequency. The clock frequency is in turn dependent on the supply voltage. If we choose to divide a data processing problem into two parts we may be able to use only half the clock frequency and still process the same amount of data in the same time period. Additionally, with a circuit running half the clock frequency, the supply voltage may be lowered as well, saving additional power. There is one catch, however. The area of two parallel processors would be about twice the size of a single processor core. Still, we see from (2.5) that the dynamic power could be lowered if the frequency and supply voltage together decreases more than the capacitance increases.

2.3.2 SIMD

Traditionally, a processor has one processing unit, and one stream of data to process. With the arrival of multicore systems, processors may now be divided into four groups based on Flynn's taxonomy [14]. Firstly, we have the traditional SISD (Single Instruction Single Data) processor, which is the single core capable of processing a single instruction at a time. The second type, MISD (Multiple Instruction Single Data) has no real-world use case, and remains a theoretical group only. The two remaining groups contain the used multiprocessor types.

These are SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data). SIMD processors share the same instructions and perform, as the name indicates, the same instructions in each core in parallel. The only difference between the cores is the data they are processing. This type of parallelism is great when having the need to process a lot of different data in the same way. For instance vector multiplication

$$c[i] = a[i] \cdot b[i]$$

would benefit from using a SIMD processor. The entire array with length L would be processed in a single instruction if the number of processing units $N \geq L$. If the array is longer than the number of processors, the processor simply starts with the N first elements and then goes on. If some of the processing elements does not have data to process however, these will be in idle mode and not computing. When processing a problem that cannot be parallelized, only one processing element will be active, and all others idle. This has a great impact on the efficiency of the system if the problem at hand cannot be processed in parallel. A simple block diagram of a SIMD processor is shown in figure 2.7 [14]. We see how all the cores have the same control logic while each processing element is working on a separate part of the memory.

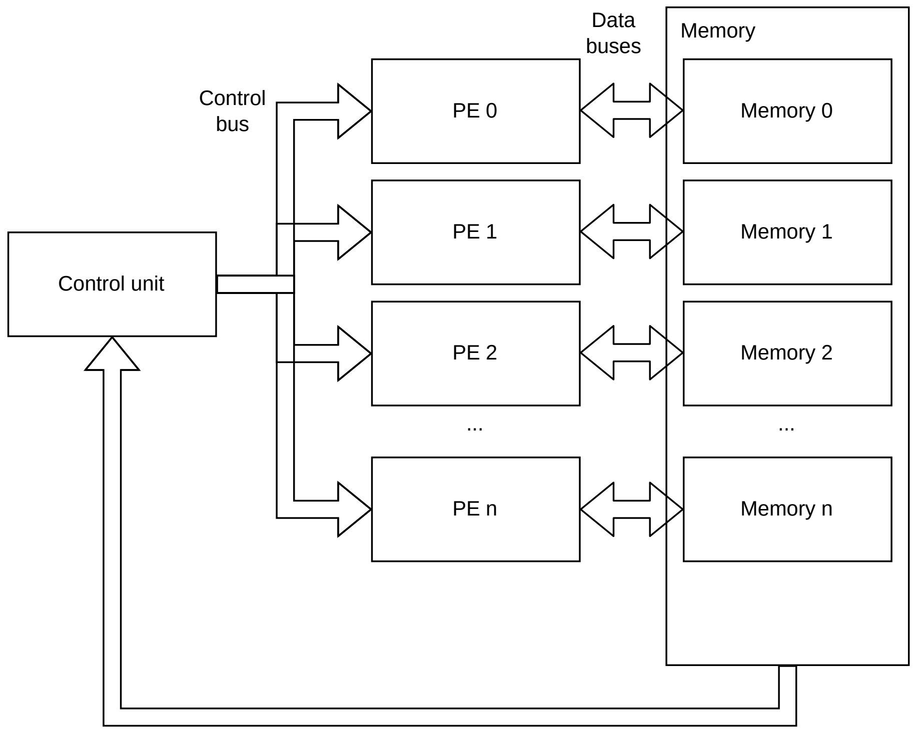

Figure 2.7: A simple overview of a SIMD processor with one control unit and several processing elements PE.

2.3.3 MIMD

MIMD processors, as SIMD processors, work simultaneously on different data, but here the instructions may differ as well. This makes MIMD processors the most flexible of the classes in Flynn's taxonomy. The cores may work on entirely different problems at the same time. The drawback with MIMD processors is the area and power consumption, as well as the troublesome programming. As each core in a MIMD processor needs its own control unit, it will not be as area or power efficient when working on problems that could be executed efficiently on SIMD processors. When working with different instructions, however, MIMD processors are more efficient [14]. A simple block overview of a MIMD processor is shown in figure 2.8.

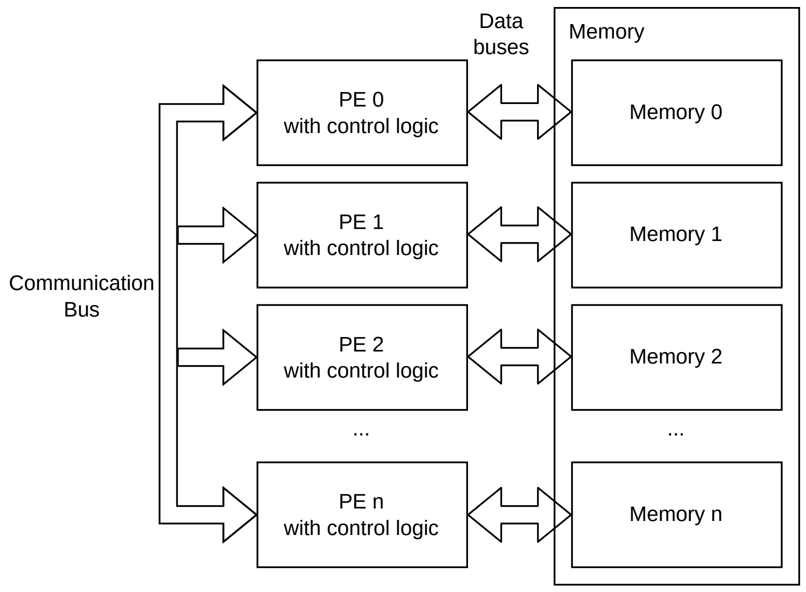

Figure 2.8: A simple distributed memory MIMD CPU with several processing elements/cores.

2.3.4 Distributed vs Shared Memory

MIMD processors may be divided further, depending on the memory system of the processor. There are two main alternatives, as well as the possibility of combining the two. These two alternatives are shared memory and distributed memory. Shared memory means there is a single block of memory that is shared between the cores of the processor. This solution ensures that all the cores have access to all the data, but it may be quite slow, at least when the number of cores grows. This is because a lot of cores will try to access the memory at the same time [14].

The second solution is distributed memory where each core has its own private memory close to the core. This solution is far more efficient if each core only needs data from its own memory location. For a core to read the memory "belonging to" another core, however, the latency will be higher as the second core will have to read it before sending it to the first core [14].

The third alternative is a combination of the two. The cores may be divided into larger groups where each group has a shared memory as well as interconnects into the other groups' memory. This is a tradeoff between latency when loading from the close-by memory, the latency when loading from far-away memory, and the hit rate of both of these.

2.3.5 Interconnects

When using MIMD processors, the cores need to have some sort of interconnect in order to synchronize and share data. There are numerous ways of designing this interconnect between the cores, depending on speed requirements, power budget, and how much data will be sent between them. A faster interconnect, like a hypercube or a direct connection between all cores are alternatives when a lot of data transfer between the cores is needed. This will on the other hand use a lot of area and power when the design has many cores. A smaller interconnect, for instance a single common bus, would have a much smaller area and power need. This, on the other hand, cannot transfer nearly as much data. As mentioned, it comes down to a tradeoff between power and area vs throughput of the interconnect [15].

2.3.6 RISC-V

RISC-V is an open-source instruction set [6]. This means that all the instructions are known to whoever wants to read them, and anyone may implement their own RISC-V based implementation free of charge. This has resulted in numerous open-source implementations as well, in different flavors. Some are simplified implementations for teaching computer architecture, some are optimized for the highest possible clock speed, and some are optimized for the smallest area.

The instruction set was developed at Berkley university, as a successor to the former RISC-IV. While the former RISC-implementations from Berkley are mostly used for research, this last version has become complete enough to use commercially as well.

One advantage of using RISC-V, besides the open-source instruction set, is the division into several subsets of the instruction set. One may choose to create an implementation with only a small subset of the instructions. The smallest subset for instance is the RISC-V32E, which only has integer operations and only 16 registers in the core [16]. This possibility to use only some of the instruction set as well as the programming tools having the same reduced instruction set possibilities makes it possible to cherry-pick only the needed instructions, leaving out the rest. By leaving out unneeded instructions from the core entirely the design might have a smaller area, resulting in a lower power dissipation as well as a lower manufacturing cost.

Beamforming Processor

As previously mentioned, beamforming takes in signals from several transducers and processes them. Then the different intermediate results are summed. This signal processing may be done on a parallel processor, either MIMD or SIMD. For the first part, before the intermediate result, each core will not need any information from other cores, which makes this type of processing ideal with a distributed memory system. The last bit where the intermediate results are summed, however, will need data from several different cores. This might not work as well with a distributed memory as it would with shared memory.

Chapter 3

Implementation

3.1 Data Rate and Power Budget

The ultrasound probe this processor design is a part of has some specific limitations and parameters as shown in table 3.1.

Table 3.1: Limitations and parameters of the ultrasound probe.

Parameter	Value
Max temperature	40°C
Max average power	3 W
Sample rate	10 MHz
Sample resolution	12 bits
Number of channels	10 000
Burst time	300 - 400µs

The first limitation is the maximum temperature of 40°C. This limitation comes from the fact that the probe will be used in medical ultrasound. Because of this limitation, the probe cannot be too hot as it would damage the skin of the patient. From the strict temperature limitation, the power limitation is set. For the probe to keep cool enough, the total power budget of the complete probe is 3W. As such a small probe in many cases would be battery powered, this also adds to the tight power budget of the probe. These 3W is the power budget for the whole probe and not only the processor. The power budget of the processor would then need to be considerably lower than these 3W.

Next, we have the vast amount of data from the transducers. The probe will have 10 000 transducers running in parallel, sending ultrasound pulses and receiving them. Each of these transducers has a data rate of 10MHz, at a 12 bits resolution. The bandwidth combined will be as follows:

$$10\ 000 channels \cdot 12 bits \cdot 10 MHz = 1.2 Tbps$$

This data rate is the new data into the processor. Additionally, a number of the last samples have to be saved for some time to be able to perform the beamforming. From (2.2) we find the maximum time delay from one transducer to the next. The exact size of the final ultrasound probe is not set, but it might be in the order of 5cm in a 100*100 transducer array. As the maximum time delay is at an angle $\theta = 0$ to the transducer plane, the equations are simplified, and we get:

$$t = \frac{0.05m \cdot cos(0)}{343m/s} = 146\mu s \approx 1500 samples$$

By limiting the probe to perform beamforming within a $\frac{\pi}{4}$ angle, the number of samples to save drastically decrease:

$$t = \frac{0.05m \cdot cos(\frac{\pi}{4})}{343m/s} = 103\mu s \approx 1000 samples$$

By implementing more efficient beamforming algorithms than using the whole array at a time, the number of samples to be saved intermediately could be lowered even more. For instance, by doing beamforming in sub-arrays first only one value from each sub-array will need to be saved before being added to the result from the other sub-arrays. As these sub-arrays are physically smaller than the whole array, each sub-array does not need as many intermediately saved samples to be able to beamform at the same angles.

To store this amount of data, a fair amount of power will be consumed. Especially with the tight power budget given, being able to process all this data could prove tricky.

3.2 PicoRV32

The open source RISC-V implementation PicoRV32 is chosen as the base for the parallel microprocessor. This implementation has a very low area per core, which is ideal when a large number of cores is needed. In order to allow this small area, the implementation is quite bare-bones. It supports the RV32IMC instruction set, yet only RV32IM is used in this multi-core implementation [16]. It has no cache and also no pipelining. It typically has about 3-6 CPI (Clock cycles Per Instruction) as a result of the lack of pipelining and cache. From an implementation on a Xilinx FPGA, its area is measured to 2123 LUTs (Look Up Table), while other RISC-V implementations in the evaluation have 13062 (Freedom) and 14616 (Pulpino RI5CY). This tiny size is the reason why this implementation is chosen [17].

The PicoRV32 processor may be divided into several submodules as seen in figure 3.1 [17]. Here we see the whole PicoRV32, connected to an external memory through a simple bus. This bus is connected to a memory interface inside the PicoRV32. It has an instruction decoder that decodes the read data when reading instructions. The decoded instruction is forwarded into the core, here called picorv32_core. This core processes data, and may also have some accelerators instantiated to speed up commonly used instructions. These accelerators are communicated with through the pico coprocessor interface pcpi. As seen in the figure, the multiplication and division accelerators are enabled in the project. The division accelerator, and the multiplication accelerator especially, are enabled because digital signal processing performs a lot of these instructions.

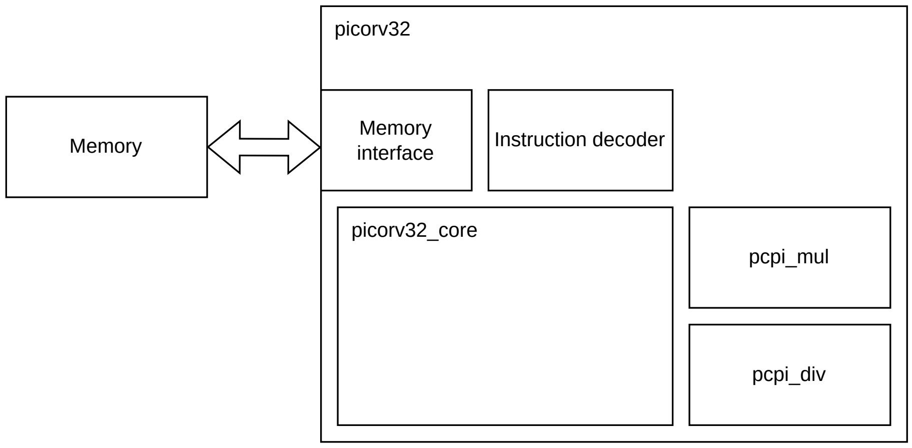

Figure 3.1: A block overview of the PicoRV32 RISC-V processor with instruction decoder, control unit and a processing element as well as accelerators for multiplication and division.

In order to process the vast 1.2Tbps of data from the ultrasound probes, more than one core is needed. As discussed in section 2.3.1 about multiprocessing, two common ways of doing multiprocessing are using MIMD or SIMD cores. As the ultrasound processing is massively parallel, and all the data will be processed in the same way, SIMD cores would work on this problem. MIMD cores, as they work independently from each other, will also work, but might add some control overhead. We will look into both approaches, and see how they compare. A block of the SIMD solution is shown in figure 3.2.

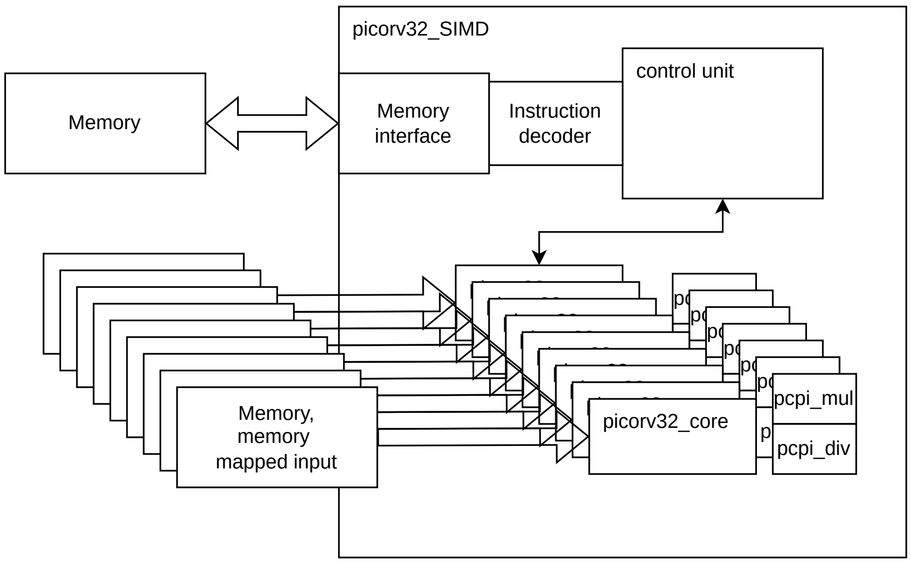

Figure 3.2: PicoRV32 modified with several SIMD cores.

The MIMD solution is shown in figure 3.3. Here we see how each MIMD processor can have several SIMD cores. They will additionally need some interconnection for synchronization and exchange of data, compared to the SIMD processor.

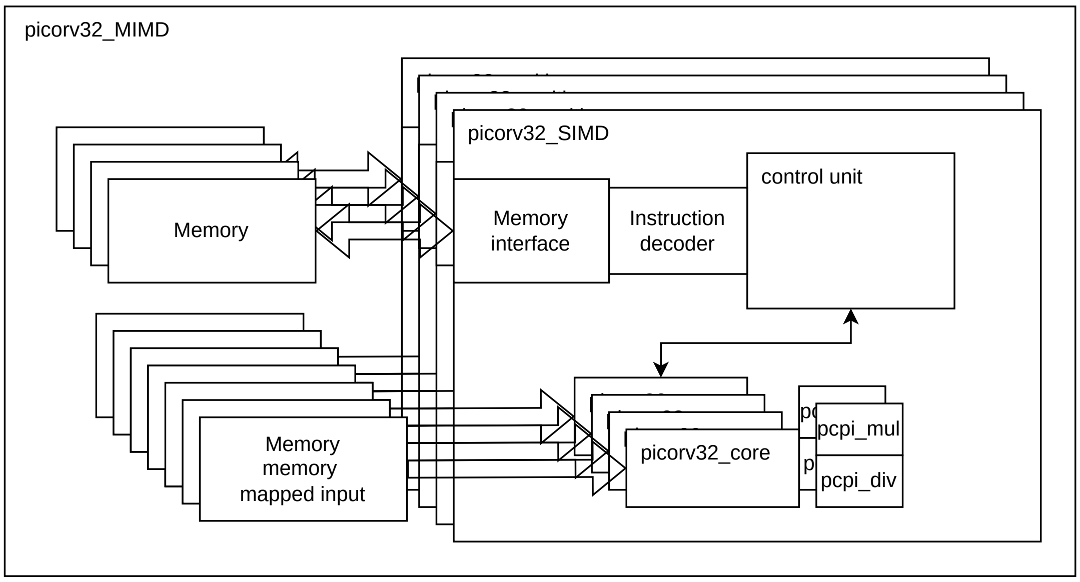

Figure 3.3: The MIMD solution of the PicoRV32. Each MIMD block may have several SIMD cores.

As we see from figure 3.3, these interconnects between the MIMD cores are not implemented, and is a part of the future work for the project. The single MIMD cores will mostly need data from one part of the memory, as one core will process data from a given number of transducers before combining the results with other cores. As the cores only read data from the transducers "belonging" to them, a distributed memory system is used, where each core will have a number of transducers each.

As there is no interconnection network as of now, the cores then have no way of combining this data that is processed in each core.

Therefore, the program for testing the implemented versions does not run a beamforming algorithm, but instead runs matrix multiplication. This is done with different data for each core, both for the SIMD and the MIMD solution.

The implemented versions are synthesized in a 22nm FDSOI technology in the SS (Slow Slow) corner running at 0.59V. This corner is used to ensure adequate slack for all corners. The power estimates are done at the TT (Typical Typical) corner at 0.65V to get the most typical power estimates.

3.3 SLVT (Super Low Voltage Threshold)

For the implementation of the design an SLVT 22nm FDSOI library is used. This might not seem the obvious choice for a low-power and relatively slow system such as this one, as we have seen from section 2.2.4. SLVT libraries are generally good for two things, speed or low supply voltage. When using an SLVT library for a relatively slow design running at a ”normal” supply voltage, the design will have a lot of leakage current.

There is one simple reason to use the SLVT library, though, there is no other option. This FDSOI technology is strictly confidential, and only the SLVT library is available for the designer at this time. In order to find out if it is possible to get reasonable estimates for dynamic power and area, a test was done using another technology, namely a 28nm FDSOI technology. The same design (the single core PicoRV32) was synthesized in both LL (Low voltage threshold) and LR (Regular voltage threshold). We see the result from this test in table 3.2.

Table 3.2: The power and area results from 28nm synthesis on the single core PicoRV32. For LL and LR.

Configuration	Dynamic Power	Static Power	Area
28nm LR	7.13µW	23.9nW	7227(µm)^2
28nm LL	7.09µW	720nW	6700(µm)^2

Here we see how the static power is a lot higher for the LR library, while the dynamic power and area are relatively similar between the two libraries. As the dynamic power and the area are so similar between LL and LR, the conclusion is drawn that it is possible to use the 22nm FDSOI SLVT library for area and dynamic power estimation, while not taking the static power consumption into account. This leakage power would be lowered when using a higher-VT library. Therefore the 22nm SLVT library is used, but only dynamic power and area are seen as good estimates.

3.4 Implemented Versions

To test different versions of the system and see how the power and area vary, many variations are implemented and tested. Three parameters are varied, as shown in table 3.3.

Table 3.3: Variables tested in the design.

Variable	Range
SIMD Cores	1, 2, 4, 8, 16, 32, 64, 128, 256
MIMD Cores	1, 2, 4, 8, 16, 32, 64, 128, 256
Clock frequency [MHz]	10, 20, 50, 100, 200, 500

As the area increases with every added core, the design soon becomes too large to synthesize in an acceptable period of time, therefore only configurations with a maximum core count of 256 cores were tested. For instance 4 MIMD cores, each consisting of 64 SIMD cores. The highest numbers of MIMD and SIMD cores are therefore not tested simultaneously.

As we will see from the results, power and area estimates for higher core count could quite easily be extrapolated from these results.

3.5 Tool Stack

3.5.1 RISC-V Toolchain

The riscv-toolchain is installed and configured to work with the riscv32im extension, as this is used in the configuration of PicoRV32. In the toolchain, gcc is used to compile C and assembly code to run on the simulated design. After creating the binary file, this is converted to hexadecimal values readable to the verilog testbench for the PicoRV32.

3.5.2 Synopsys VCS

Synopsys VCS (Verilog Compiler and Simulator) is used for simulation and verification of the design. It compiles RTL (Register Transfer Level) Verilog code, and together with a testbench it simulates and verifies the system. In this particular case, with the design being a processor, the verification is done by running a compiled C-code on the simulated processor as discussed above. This is an effective way of creating stimuli for the design compared to manually setting input values. Additionally, this makes it possible for the core to print out messages through VCS, making it easier to see what the core is doing than looking into the waveforms. VCS may additionally dump waveforms to file, so that the power estimation of the system may become more accurate as discussed in section 3.5.5 later [18].

3.5.3 GTKWave

GTKWave is used in the debugging process. It reads waveform files created by VCS, so that all the signals inside the design are visible to the designer. This can help to discover the smallest problems, but might also be too full of details as the information base when dealing with a whole CPU is huge. One neat feature is the ability to show bus signals as hexadecimal values instead of bits, which makes them a lot more readable [19].

3.5.4 Synopsys DC

Synopsys DC (Design Compiler) compiles the verilog RTL code to a single gate level verilog file. This is done by using a library file for the 22nm FDSOI technology. As this technology file has the physical implementation of all the blocks used in the design, the first physical estimates are found when synthesizing the design using Synopsys DC. These estimates include an estimate for the power, timing and area among others. Yet DC does not know anything about the activity of the design when operating, it just adds a default switching of every input net. This makes the power estimate somewhat less accurate than what could be possible using an activity file. DC performs some optimization to the design as well, like removing registers that do not drive any nets and removing other unused parts of the design. DC also inserts clock gating cells, stopping the clock from propagating to parts of the design that do not need the clock signal. These modifications could sometimes result in some alterations to the function of the circuit. Therefore a post-synthesis simulation is also in place. That is, running VCS again, but using the gate level netlist instead of the RTL [20].

3.5.5 Synopsys PT

Synopsys PT (PrimeTime) is used to get more accurate estimates for the timing and power consumption of the circuit. PT runs on the synthesized netlist, combined with the library files containing the physical characteristics of the cells. By adding a waveforms file from the VCD run, the power estimates become more accurate for those scenarios run in the simulation. Here it is also possible to run several scenarios, for example one where the design is running as fast as possible, and another where it is sleeping. Then PrimeTime can estimate the power consumption in the scenarios, giving a better overview of the different run modes [21].

Chapter 4

Results and Discussion

4.1 Area

The first results we will look into are the area estimates. In figure 4.1 we see how the area is approximately linear to the number of cores in the design. In the figure, the number of SIMD cores is at the x-axis, with a different line for each number of MIMD cores. As the clock frequency does not much alter the size of the design, only the area for the 10MHz implementations are shown. All results are shown in appendix A.

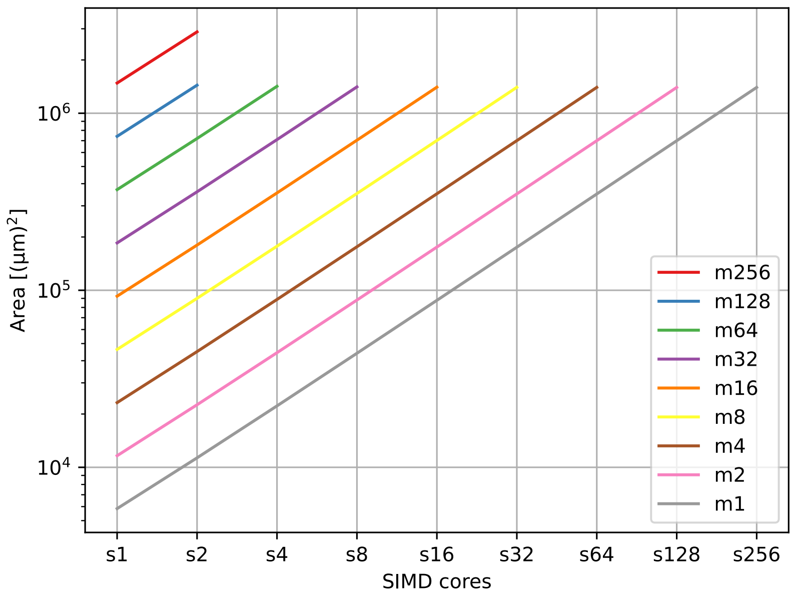

Figure 4.1: Area per MIMD and SIMD core.

The results only contain estimates for when the total number of cores is a maximum of 256, with one exception with 256 MIMD cores, each with 2 SIMD cores. The reason for this limitation at 256 cores, is the synthesis time. As the size of the design increases, so does the duration of the synthesis. As the area is very close to linear, it seems that in order to get an initial estimate of a design with a greater number of cores, one may extrapolate the data from these results.

From figure 4.1 it looks like the size of the design does not differ much between using SIMD cores or MIMD cores. As the SIMD cores share one instruction decoder between them while MIMD cores have one each, it would be natural for the SIMD cores to be somewhat smaller than the MIMD cores. This small difference is seen in figure 4.2.

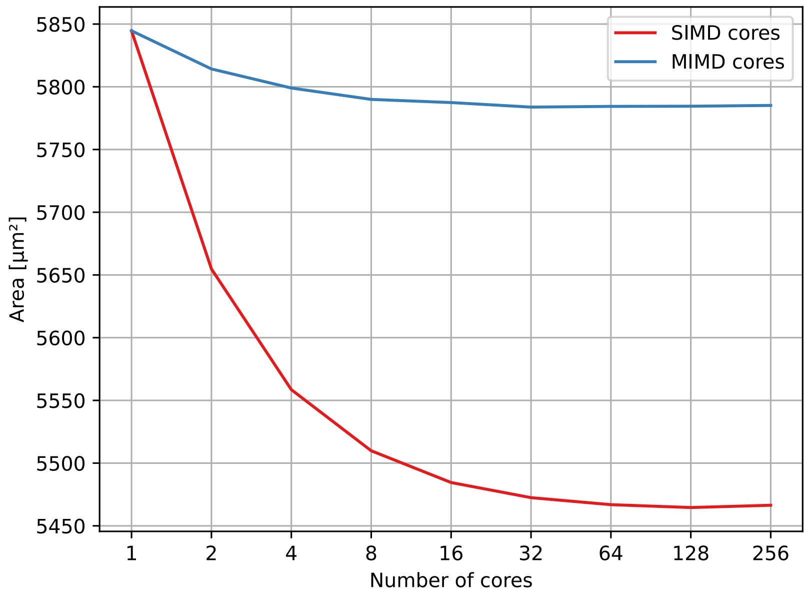

Figure 4.2: The average area for each core depending on the number of cores in the design.

We see here how the area per core is smaller for the SIMD cores than for the MIMD cores when the number of cores gets higher. As mentioned, this is because the SIMD cores share the instruction decoder between themselves, whereas the MIMD cores have one each. The MIMD cores also reduce slightly in size when the number increases. There might be some small part of the core that can be shared between them as well, for instance some of the input ports to the design. The reduction in average size is much bigger for the SIMD cores, as expected. The results in the figure are again estimated at a clock frequency of 10MHz.

Figure 4.3 shows the area per core over frequency. Only the SIMD cores are shown, as these vary much more than the MIMD cores. We see how the area per core stays almost the same for each number of cores, but increases when the clock frequency is set to 500MHz. This shows how the synthesized design is roughly the same for all the lower frequencies.

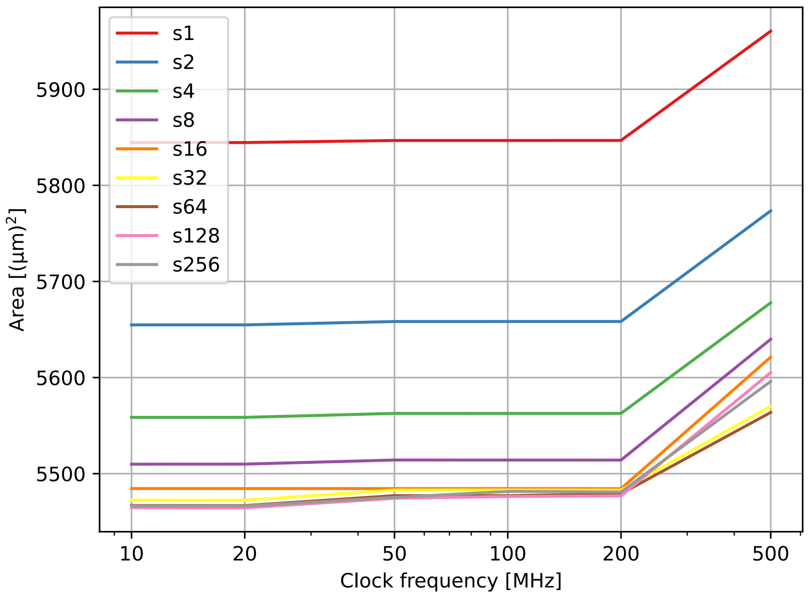

Figure 4.3: Area per SIMD core as a function of clock frequency.

When the clock frequency reaches 500MHz the design starts to change. This might seem strange, but has a natural explanation. This is namely where we start to have timing problems. We will discuss this more in the next section, but for lower frequencies we have no timing problems whatsoever, while at 500MHz the initial synthesized design has negative slack. The synthesis tool then changes the netlist to use other blocks in order to further speed up the design. These blocks are larger than what is needed at the lower clock frequencies, resulting in a larger area at 500MHz.

4.2 Slack

As mentioned, timing issues start at around 500MHz. In figure 4.4 we see how the slack is almost as high as the clock period for the lower frequencies. At around 100MHz we see that the slack is closer to half the clock period, and at 200MHz the designs with the highest number of cores become dangerously close to zero slack.

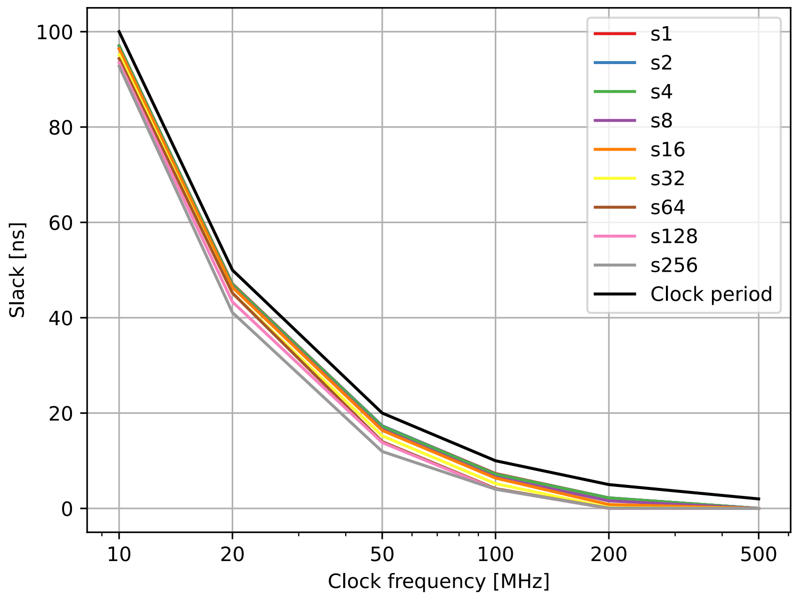

Figure 4.4: Slack as a function of clock frequency for SIMD cores.

In figure 4.5 we see more clearly what happens, especially at higher frequencies. This figure shows the slack as a factor of the clock period.

Here we see how at low frequencies the slack is almost the whole clock period. The slack becomes a smaller and smaller part of the clock period until it drops almost to zero at 200MHz for the designs with more cores. At 500MHz it is close to zero for any number of SIMD cores. Yet as discussed in the area section, the synthesis tool handles this and creates a larger design in order to fix the timing problems. We have seen in figure 4.3 how the area increases at the highest frequencies as a result of this.

An even higher frequency would make the design area even larger, but there is a limit to what the synthesis tool can do. Eventually, the frequency would become too high for the design in this technology, and changes would have to be made either to the design or in the choice of technology in order to further speed up the clock.

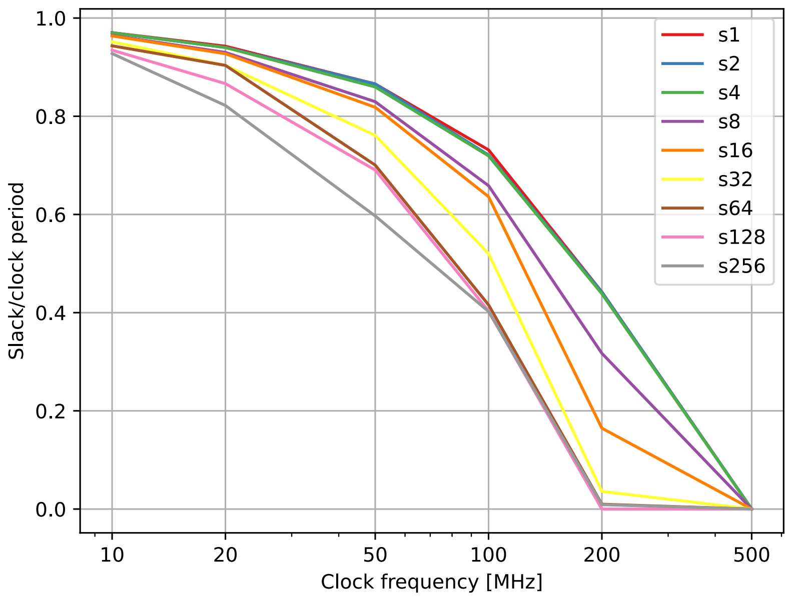

Figure 4.5: Slack as a factor of the clock period for SIMD cores, as a function of the clock frequency.

4.3 Power

As mentioned in section 2.2.4 about voltage threshold, only the SLVT (Super Low Voltage Threshold) libraries are available for this design. This results in a faster circuit, yet with a lot more leakage current. Therefore, as mentioned, the dynamic power will be the main point of interest, while the leakage not so much.

As a 28nm FDSOI technology library was also available, a small test was done, showing that the dynamic power stays approximately the same between different voltage threshold libraries in 28nm, while the leakage power is very much different. This assumption that the dynamic power stays mostly the same across different voltage thresholds is why it is believed that the dynamic power results in 22nm are still good estimates even when using a ”wrong” voltage threshold library for this design, especially at the lowest clock frequencies.

Figure 4.6 shows the dynamic power divided by the number of cores for MIMD and SIMD processors. Much like the area per core, we see here how the SIMD cores do not need as much power as they share the instruction decoder.

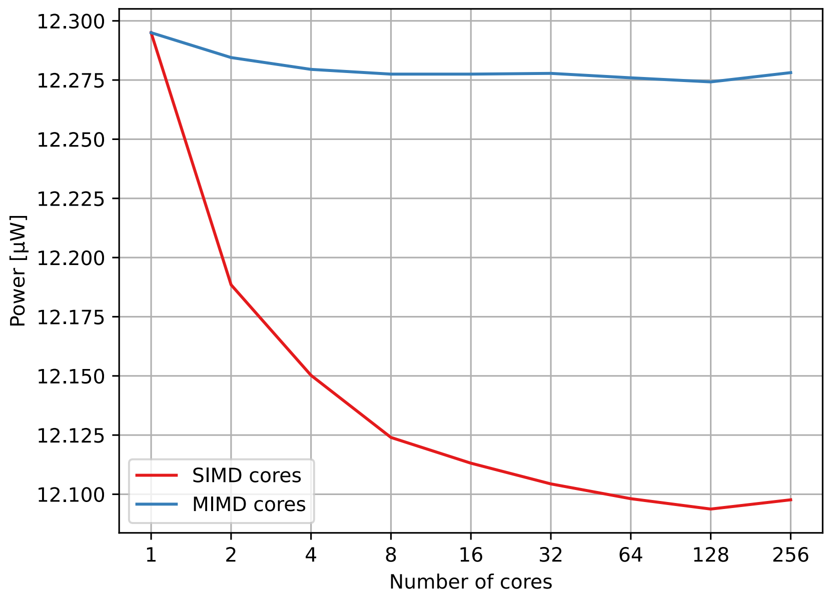

Figure 4.6: Dynamic power divided by the number of cores.

One curious observation is the fact that when going from 128 cores to 256, the power per core grows for both MIMD and SIMD cores. The difference is small, but still there. One reason might be the longer wires needed in such a large design. When the number of cores increases, the design uses a larger area, and the common nets between the cores become longer. These are nets from the instruction decoder for the SIMD cores, and also nets such as the clock.

Another reason could be that the synthesis tool does not perform as much optimization. As the synthesis time becomes larger with every core, the synthesis time for 256 cores might be so long that it skips some of the fine-tuning done when fewer cores are instantiated to save time, and therefore ends up with a slightly less power efficient design.

In figure 4.7 we see how the dynamic power rises with a higher clock frequency. From (2.5) we know that the dynamic power is proportional to the clock frequency of the system and we see this connection in these results as well. This figure only shows a variation of SIMD cores, as we have seen from figure 4.6 how the SIMD and MIMD based designs are very close in power consumption.

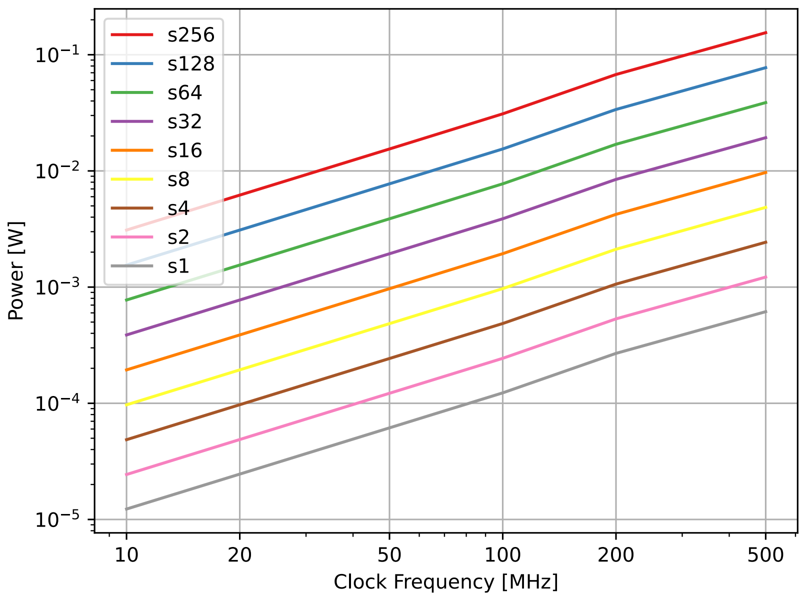

Figure 4.7: Dynamic power as a function of frequency for SIMD cores.

In figure 4.8 we see the power consumption of a single core divided into the dynamic power and the static leakage power. The total power is also shown.

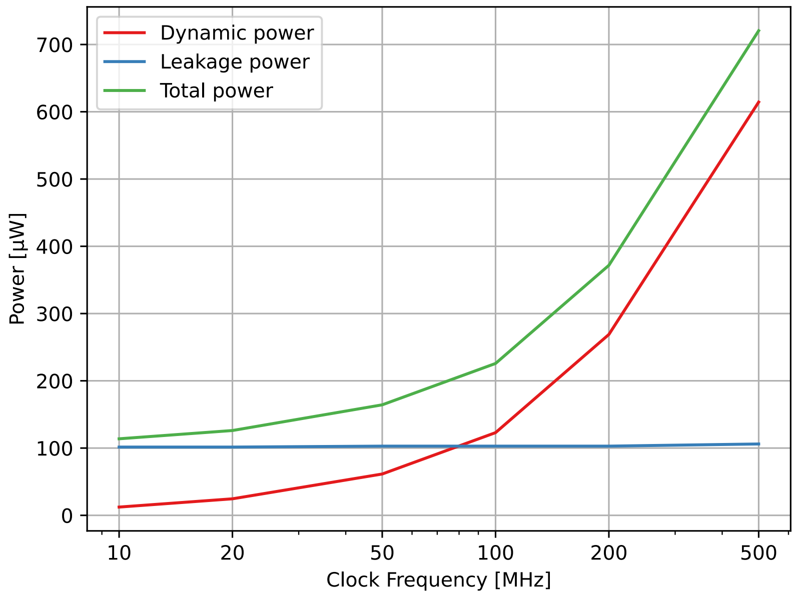

Figure 4.8: Power per frequency for one core divided into dynamic and static power.

From this figure we see why the SLVT library is not the best library to use with this design at low frequencies. That is, at 10MHz the leakage power contributes to about 90% of the total power consumption. As mentioned in section 2.2.1, the best tradeoff between speed and leakage for a design is where the leakage power is in the same range as the dynamic power. Running this design at 10MHz, the SLVT is not a good fit. The SLVT is a fast, but leaky, library. A slower library with less leakage would fit much better for these slow clock frequencies.

When the clock frequency increases, on the other hand, the dynamic power consumption increases much more than the static power. The static power stays almost the same over frequency, as found in (2.6). At a frequency slightly below 100MHz, the dynamic power outgrows the static power, as it continues to rise with the frequency. This high-frequency area is where the SLVT, as a fast technology, has its rightful place. Here the dynamic power is what contributes most to the total power, and not the leakage current just wasting energy. As the slack is still positive, but just barely, at 500MHz, this might be around the highest frequency this design can run with this specific SLVT library. A mix of libraries could however lower the power consumption further, using the SLVT library where speed is of utmost importance while using slower libraries in parts of the design where high speed is not needed.

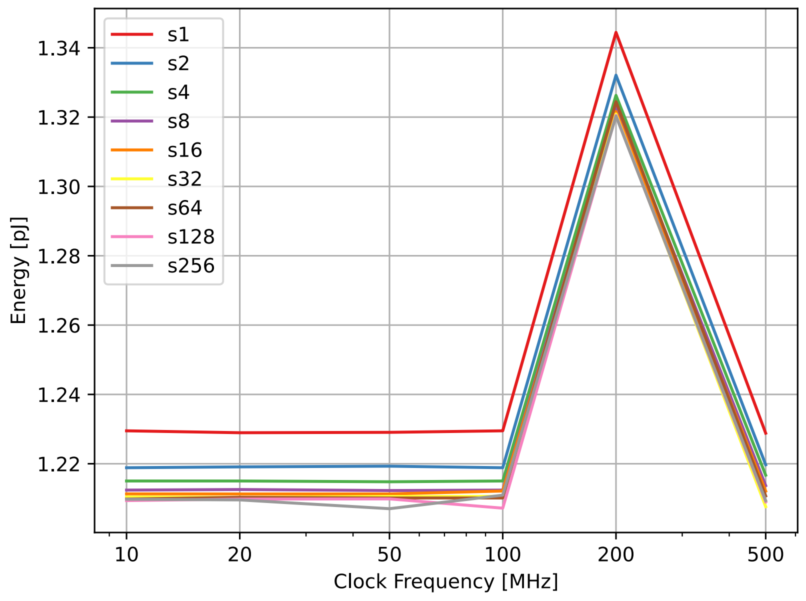

Figure 4.9: Dynamic energy per clock cycle divided by the number of cores for the different number of SIMD cores and clock frequency.

In figure 4.9 we see the dynamic energy of a single clock cycle divided by the number of cores. Here we see how the dynamic energy is approximately the same per cycle independent of clock frequency. We also see how the designs with more cores tend to use less energy per cycle per core. As we have seen from figure 4.6, the dynamic energy per core is less when many cores are instantiated, as they share some of the same logic. Something seems to happen at 200MHz. Here the dynamic energy is estimated to be quite a bit higher than at the other frequencies. Why this happens remains a mystery, but a couple of guesses could be made. Firstly, it could be some sort of resonance frequency in some nets, resulting in a higher voltage swing, and thus a higher energy consumption. Alternatively, it could be the power estimation tool having trouble at this exact frequency for some obscure reason, which gives this result. Either way, it is believed to be a mistake in one way or another.

If we instead take look at the total energy divided by the number of cores per cycle, we get figure 4.10. Again we see how multiple cores tend to consume a bit less energy per core than fewer cores. Across frequency, however, the energy varies much more. As the leakage power stays about the same no matter the clock frequency, more energy is consumed per cycle at a lower clock frequency. From this figure, a higher clock frequency is definitely more energy efficient.

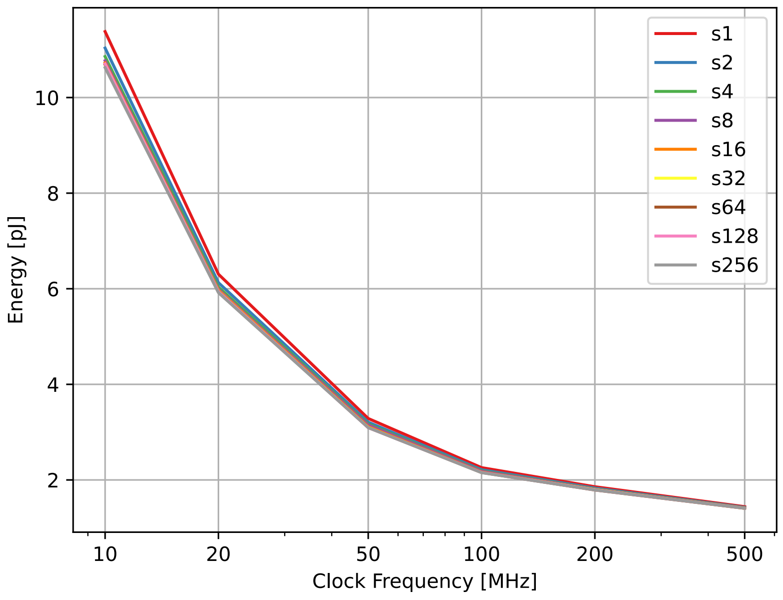

Figure 4.10: Total energy per clock cycle for different number of SIMD cores and frequency.

By changing the used library to a library with a higher voltage threshold the leakage current could be reduced for the lower frequencies, and the figure might have had a flatter graph. Another possibility is to lower the supply voltage of the entire design. The slack is high at the low frequencies and the leakage power is high, so lowering the supply voltage would be possible without timing problems.

4.4 Further Discussion

For the timing results, the design is run at 0.59V in the SS corner and the power at 0.65V in the TT corner. As the slack is way above what is needed for the slow frequencies, the voltage could be lowered further. Unfortunately, this is the lowest voltage available for the technology. With the slack at 10MHz, it could be possible to run the design at voltages as low as 0.3 or 0.4V. This would significantly lower both the dynamic and static power consumption, while still doing all the same work in the same time period.

The CPI (Clocks Per Instruction) is measured to be 6.9 on average per core when running matrix multiplication. This includes both reading the instruction, fetching data, executing and writing back the results. However as the matrix multiplication is rather different than a beamforming algorithm, it is hard to say anything about the frequency and the number of cores needed to perform beamforming.

As mentioned, the design is tested by running matrix multiplication and is not capable of running a multicore beamforming algorithm in its current state. To be able to do beamforming if will need interconnects between the cores for sending data and synchronization. The memory setup is a distributed memory and the data from one core cannot be read by another. By implementing a network for communication between the cores it would be possible to exchange the needed data. Another solution could be a shared memory where all the cores (or cores belonging to a group) could all read and write, but this solution introduces new problems with read and write latency, as well as issues with data consistency.

In the design, only the actual processor is synthesized and used for power estimation. The memory is currently a part of the testbench and does not contribute to the power or area estimates. By adding memory and also the needed interconnection between the cores, both the power and area estimates will increase. Yet it makes no sense to implement the memory in this fast SLVT library, as the implementation of a low-area low-energy memory is a problem in itself, and would not be a good match for this library. The memory is therefore left out of the power and area estimates.

A clean way of reading input data from the transducers is mapping them to addresses in the memory, so-called memory-mapped I/O. This makes the input data "look like" any other data in the memory, except it is read-only. The cores will read from the memory address belonging to the transducer it needs, and get the sample from the transducer. A FIFO (First In, First Out) array could be mapped into the memory, so that the N last samples are available to the processor. This way, reading a delayed sample will be an easy task.

A simple but powerful improvement to the cores would be the ability to vectorize the data into two 16 bit parts. As the transducer data is only 12 bits wide while being processed on a 32-bit processor, the high bits are not used in the calculations. An improvement would be the ability to process two values at the same time. One sample would use the lower 16 bits, while the other uses the high 16 bits. This would double the throughput without any major changes to the core, processing two samples at the time period previously used to only process one.

A final, yet important possibility is sleep mode for the processor. As the transducers have a burst period of 300-400µs, the processor could be disabled and power gated between these bursts. This would reduce the leakage power significantly in these periods and thus contribute to a lower total power consumption. As it takes some time to wake from sleep mode, the periods where the processor is in sleep mode would have to be significantly long in order to save energy. If not, the extra power used to put it into and out of sleep mode would be more than the saved energy. If the period between the burst is not long enough, the time when the probe is on while not scanning a patient certainly is. In these periods between ultrasound scans when the image processing is not needed at all, the processor should have a sleep mode enabled to save power.

Chapter 5

Conclusion

In this project, the PicoRV32, a low area RISC-V-based processor, has been modified to have several SIMD cores divided into one or more MIMD cores. The processor is created with processing ultrasound transducer data from 10 000 transducers in parallel at a 10MHz sample rate in mind. As the processor will be internally in the ultrasound probe, there are strict temperature and power limitations at 40°C and 3W respectively for the whole probe.

In order to do this processing, a SIMD version and a MIMD version has been implemented and tested for 1 to 256 cores each, as well as clock frequency from 10MHz to 500MHz. As no interconnection between the cores is implemented, the test program runs matrix multiplication. The area estimates are approximately 5500µm^2 for each SIMD core and 5800µm^2 for the MIMD cores due to the fact that the SIMD cores share some parts of the logic.

The SIMD cores have a dynamic power dissipation of approximately 12.1µW per core while dissipating almost 12.3µW per core for the MIMD cores at 10MHz. At 500MHz the dynamic power dissipation is 605µW per SIMD core and 613µW for each MIMD core.

As there is a lot of slack in the design, a higher voltage threshold or a lower supply voltage would be beneficial in order to save more power. Still, as the power dissipation for the design is relatively small compared to the 3W power budget for the entire probe it seems such a system could be useful in the next generation ultrasound probes.

5.1 Future Work

First and foremost, the design needs a NoC (Network on Chip) to be able to communicate between the cores. Without this, the different cores cannot pass data between them to perform a beamforming algorithm. Beamforming is dependent on data from several of the cores, and this is needed to pass the data.

With the NoC in place, the next improvement would be for the processor to perform an actual beamforming algorithm. This would give us better results of how much power is needed for this kind of computation. From these results, it would be possible to draw a conclusion as to how many cores are needed in such a system in order to perform the beamforming algorithm in time.

Another improvement would be to try and synthesize the design at a high voltage threshold. For the lower range of the tested frequencies, a high threshold library would give much better estimates for the static power consumption of the design. And for the higher frequency tests as well, it would be interesting to see how much of the design actually needs the speed an SLVT library brings, and how much of it could run just as good at a high voltage threshold library.

Sleep mode would also help drastically on the power consumption, and should absolutely be implemented into the design. After implementing a sleep mode it would be interesting to see how much power is saved by going into sleep. This would give a much more accurate estimate of how much the final power dissipation of the design would be.

Finally, the memory block would have to be implemented into the design. When the memory block is also present in the design, the final estimates for power and area could be found. After adding the memory into the design, further tweaks of the design would be in the software running on the processor.

Bibliography

[1] Libertario Demi. Practical guide to ultrasound beam forming: Beam pattern and image reconstruction analysis. *Applied sciences*, 8(9):1544, 2018. ISSN 2076-3417.

[2] Murtaza Ali, Dave Magee, and Udayan Dasgupta. Signal processing overview of ultrasound systems for medical imaging. *SPRAB12, Texas Instruments, Texas*, 2008.

[3] Holger Hewener, Christoph Risser, Selina Barry-Hummel, Heinrich Fonfara, Marc Fournelle, and Steffen Tretbar. Integrated 1024 channel ultrasound beamformer for ultrasound research. In *2020 IEEE International Ultrasonics Symposium (IUS)*, pages 1–4, 2020. doi: 10.1109/IUS46767.2020.9251700.

[4] Claire Doody, Hazel Starritt, and Francis Duck. Prediction of the temperature rise at the surface of clinical ultrasound transducers. *BMUS Bulletin*, 11(3): 26–28, 2003. doi: 10.1177/1742271X0301100307. URL `https://doi.org/10.1177/1742271X0301100307`.

[5] Yanick Baribeau, Aidan Sharkey, Omar Chaudhary, Santiago Krumm, Huma Fatima, Feroze Mahmood, and Robina Matyal. Handheld point-of-care ultrasound probes: The new generation of pocus. *Journal of cardiothoracic and vascular anesthesia*, 34(11):3139–3145, 2020. ISSN 1053-0770.

[6] 2022. URL `https://riscv.org/`.

[7] Jan Gray. Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–20, 2016. doi: 10.1109/FCCM.2016.12.

[8] Jacob Benesty. Fundamentals of differential beamforming, 2016.

[9] John G. Proakis and Dimitris G. Monolakis. *Digital signal processing*, chapter 1. Pearson Prentice Hall, Upper Saddle River, N.J, 4th ed. edition, 2007. ISBN 0131873741.

[10] Neil H.E Weste and David M. Harris. *Integrated circuit design*. Pearson, Boston, Mass, 4th ed. edition, 2011. ISBN 9780321696946.

[11] Paul Horowitz. *The art of electronics*, chapter 10. Cambridge University Press, New York, 3rd ed. edition, 2015. ISBN 9780521809269.

[12] A Amara and P Royannez. Vhdl for low power. In C Piguet, editor, *Low-Power CMOS Circuits: Technology, Logic Design and CAD Tools*, chapter 11. CRC Press, 1st ed. edition, 2006. doi: https://doi.org/10.1201/9781315220710.

[13] Kangguo Cheng and Ali Khakifirooz. Fully depleted soi (fdsoi) technology. *Science China. Information sciences*, 59(6):1–15, 2016. ISSN 1674-733X.

[14] Peter S Pacheco. An introduction to parallel programming, 2011.

[15] L. Ciminiera, C. Demartini, and A. Serra. Interconnection networks for mimd machines. In Jürg D. Becker and Ignaz Eisele, editors, *WOPPLOT 83 Parallel Processing: Logic, Organization, and Technology*, pages 110–131, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg. ISBN 978-3-540-38803-6.

[16] 2019. URL `https://riscv.org/technical/specifications/`.

[17] Roland Höller, Dominic Haselberger, Dominik Ballek, Peter Rössler, Markus Krapfenbauer, and Martin Linauer. Open-source risc-v processor ip cores for fpgas — overview and evaluation. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–6, 2019. doi: 10.1109/MECO.2019.8760205.

[18] 2022. URL `https://www.synopsys.com/verification/simulation/vcs.html`.

[19] 2022. URL `http://gtkwave.sourceforge.net/`.

[20] 2022. URL `https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html`.

[21] 2022. URL `https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html`.

Appendix A

Table of All Data

Table A.1: Table of all configurations with results.

Configuration	Design Area µM²	Leaf Cell Count	Critical Path Slack ps	Net Switching Power µW	Cell Internal Power µW	Cell Leakage Power µW	Total Power µW
S1_M1_ck10MHz	5845	8383	97018	1.135	11.16	101.5	113.8
S1_M1_ck20MHz	5845	8383	47132	2.269	22.31	101.5	126.1
S1_M1_ck50MHz	5847	8351	17315	5.673	55.78	102.9	164.3
S1_M1_ck100MHz	5847	8351	7315	11.35	111.6	102.9	225.8
S1_M1_ck200MHz	5847	8352	2210	22.69	246.2	102.9	371.7
S1_M1_ck500MHz	5961	8635	0.12	56.91	557.5	106.1	720.5
S1_M2_ck10MHz	11628	16737	96901	2.269	22.3	202.3	226.9
S1_M2_ck20MHz	11628	16739	46901	4.538	44.6	202.4	251.5
S1_M2_ck50MHz	11633	16668	17090	11.34	111.5	205.0	327.8
S1_M2_ck100MHz	11633	16669	7090	22.69	223.0	205.0	450.7
S1_M2_ck200MHz	11633	16668	2090	45.38	492.1	205.0	742.4
S1_M2_ck500MHz	11861	17202	0.12	113.8	1115	211.1	1440
S1_M4_ck10MHz	23196	33445	96901	4.538	44.58	404.0	453.1
S1_M4_ck20MHz	23196	33449	46901	9.075	89.16	404.0	502.2
S1_M4_ck50MHz	23206	33302	17090	22.69	222.9	409.2	654.9
S1_M4_ck100MHz	23206	33302	7090	45.38	445.8	409.2	900.4
S1_M4_ck200MHz	23206	33303	2090	90.75	983.8	409.2	1484
S1_M4_ck500MHz	23674	34285	0.44	227.7	2231	421.0	2879
S1_M8_ck10MHz	46320	66734	96901	9.07	89.15	808.0	906.2
S1_M8_ck20MHz	46320	66742	46901	18.14	178.3	808.0	1003
S1_M8_ck50MHz	46339	66444	17090	45.35	445.8	818.5	1310
S1_M8_ck100MHz	46339	66444	7090	90.7	891.5	818.5	1801
S1_M8_ck200MHz	46340	66446	2090	181.4	1967	818.5	2967
S1_M8_ck500MHz	47366	68535	0.31	454.9	4460	844.5	5760
S1_M16_ck10MHz	92599	133584	96897	18.14	178.3	1616	1813
S1_M16_ck20MHz	92599	133600	46897	36.28	356.6	1617	2009
S1_M16_ck50MHz	92639	132994	17089	90.69	891.5	1637	2620
S1_M16_ck100MHz	92638	132994	7090	181.4	1783	1637	3602
S1_M16_ck200MHz	92638	132993	2090	362.8	3934	1637	5935
S1_M16_ck500MHz	94759	136821	2.89	909.9	8921	1690	11500
S1_M32_ck10MHz	185082	266935	96897	36.29	356.6	3229	3621

Configuration	Design Area µM^2	Leaf Cell Count	Critical Path Slack ps	Net Switching Power µW	Cell Internal Power µW	Cell Leakage Power µW	Total Power µW
S1_M32_ck20MHz	185082	266966	46897	72.59	713.1	3228	4014
S1_M32_ck50MHz	185152	265895	17088	181.5	1783	3270	5235
S1_M32_ck100MHz	185152	265897	7089	362.9	3566	3270	7199
S1_M32_ck200MHz	185152	265896	2088	725.9	7868	3270	11900
S1_M32_ck500MHz	188797	275014	0.0	1823	17800	3360	23000
S1_M64_ck10MHz	370200	534272	96897	72.56	713.1	6463	7249
S1_M64_ck20MHz	370203	534350	46897	145.1	1426	6464	8035
S1_M64_ck50MHz	370342	532239	17085	363.0	3567	6547	10500
S1_M64_ck100MHz	370340	532233	7085	725.6	7132	6547	14400
S1_M64_ck200MHz	370343	532242	2085	1451	15699	6547	23700
S1_M64_ck500MHz	377771	550989	0.01	3639	35600	6727	46000
S1_M128_ck10MHz	740416	1068406	96900	145.1	1426	12900	14500
S1_M128_ck20MHz	740419	1068554	46897	290.2	2853	12900	16100
S1_M128_ck50MHz	740413	1068517	16897	725.5	7131	12900	20800
S1_M128_ck100MHz	740413	1068518	6897	1451	14300	12900	28600
S1_M128_ck200MHz	740418	1068546	1897	2902	31500	12900	47300
S1_M128_ck500MHz	755243	1100904	0.03	7282	71300	13400	92000
S1_M256_ck10MHz	1480993	2138320	97173	290.2	2853	25900	29000
S1_M256_ck20MHz	1480996	2138336	47170	580.4	5705	25900	32099
S1_M256_ck50MHz	1481010	2138378	17132	1451	14300	25900	41600
S1_M256_ck100MHz	1481006	2138363	7132	2902	28500	25900	57300
S1_M256_ck200MHz	1481191	2138746	2132	5804	62900	25900	94600
S1_M256_ck500MHz	1515290	2219875	0.0	14600	142400	27000	183900
S2_M1_ck10MHz	11310	16260	97007	2.267	22.11	196.4	220.7
S2_M1_ck20MHz	11310	16262	47007	4.534	44.23	196.4	245.1
S2_M1_ck50MHz	11317	16157	17315	11.33	110.6	199.0	320.9
S2_M1_ck100MHz	11317	16158	7210	22.67	221.1	199.0	442.8
S2_M1_ck200MHz	11317	16158	2210	45.34	487.5	199.0	731.9
S2_M1_ck500MHz	11547	16817	0.3	113.7	1106	205.5	1426
S2_M2_ck10MHz	22558	32489	96901	4.533	44.22	392.0	440.7
S2_M2_ck20MHz	22558	32495	46901	9.067	88.43	392.0	489.5
S2_M2_ck50MHz	22573	32278	17090	22.67	221.1	397.2	641.0
S2_M2_ck100MHz	22573	32279	7090	45.33	442.2	397.3	884.8

Configuration	Design Area µM^2	Leaf Cell Count	Critical Path Slack ps	Net Switching Power µW	Cell Internal Power µW	Cell Leakage Power µW	Total Power µW
S2_M2_ck200MHz	22573	32279	2090	90.67	974.8	397.2	1463
S2_M2_ck500MHz	23032	33444	0.01	227.4	2212	409.5	2849
S2_M4_ck10MHz	45045	64821	96901	9.065	88.43	783.1	880.6
S2_M4_ck20MHz	45045	64828	46901	18.13	176.9	783.1	978.1
S2_M4_ck50MHz	45074	64394	17090	45.33	442.1	793.7	1281
S2_M4_ck100MHz	45074	64397	7090	90.76	884.8	793.7	1769
S2_M4_ck200MHz	45074	64396	2090	181.3	1949	793.6	2924
S2_M4_ck500MHz	46082	66964	1.26	454.6	4424	820.8	5700
S2_M8_ck10MHz	90032	129731	96897	18.13	176.8	1566	1761
S2_M8_ck20MHz	90032	129749	46897	36.26	353.7	1566	1955
S2_M8_ck50MHz	90089	128890	17090	90.64	884.2	1587	2562
S2_M8_ck100MHz	90089	128888	7090	181.3	1768	1587	3537
S2_M8_ck200MHz	90089	128888	2090	362.6	3898	1587	5848
S2_M8_ck500MHz	91973	134184	0.08	909.0	8846	1637	11400
S2_M16_ck10MHz	179983	259137	96897	36.26	353.7	3130	3520
S2_M16_ck20MHz	179983	259173	46897	72.52	707.4	3130	3909
S2_M16_ck50MHz	180089	257612	17090	181.3	1768	3172	5122
S2_M16_ck100MHz	180088	257612	7090	362.5	3537	3172	7071
S2_M16_ck200MHz	180089	257613	2090	725.2	7797	3172	11700
S2_M16_ck500MHz	183120	265226	0.05	1819	17700	3257	22700
S2_M32_ck10MHz	359931	518469	96897	72.54	707.3	6260	7040
S2_M32_ck20MHz	359931	518534	46897	145.1	1415	6260	7820
S2_M32_ck50MHz	360143	515367	17085	362.8	3537	6345	10200
S2_M32_ck100MHz	360143	515366	7085	725.3	7073	6345	14100
S2_M32_ck200MHz	360143	515370	2085	1451	15600	6344	23400
S2_M32_ck500MHz	366089	529812	0.02	3638	35300	6506	45500
S2_M64_ck10MHz	719898	1037549	96900	145.0	1415	12500	14100
S2_M64_ck20MHz	719896	1037680	46900	290.1	2829	12500	15600
S2_M64_ck50MHz	719895	1037696	16897	725.2	7074	12500	20300
S2_M64_ck100MHz	719901	1037738	6897	1450	14100	12500	28100
S2_M64_ck200MHz	719903	1037746	1897	2901	31200	12500	46600
S2_M64_ck500MHz	733745	1067065	0.0	7277	70700	13000	91000
S2_M128_ck10MHz	1440043	2075278	97126	290.0	2829	25100	28200

Configuration	Design Area μM^2	Leaf Cell Count	Critical Path Slack ps	Net Switching Power µW	Cell Internal Power µW	Cell Leakage Power µW	Total Power µW
S2_M128_ck20MHz	1440049	2075416	47126	580.1	5659	25100	31300
S2_M128_ck50MHz	1440049	2075421	17126	1450	14100	25100	40700
S2_M128_ck100MHz	1440056	2075453	7126	2900	28300	25100	56300
S2_M128_ck200MHz	1440046	2075433	2126	5802	62400	25100	93200
S2_M128_ck500MHz	1472884	2143493	0.0	14500	141300	26200	182000
S2_M256_ck10MHz	2881241	4159350	97126	580.1	5658	50100	56400
S4_M1_ck10MHz	22234	31910	97004	4.551	44.05	385.7	434.3
S4_M1_ck20MHz	22234	31915	47004	9.102	88.1	385.7	482.9
S4_M1_ck50MHz	22251	31677	17193	22.76	220.2	391.0	634.0
S4_M1_ck100MHz	22251	31676	7193	45.51	440.5	391.0	877.0
S4_M1_ck200MHz	22251	31676	2193	91.02	970.0	391.0	1452
S4_M1_ck500MHz	22711	33223	0.06	228.4	2205	403.7	2837
S4_M2_ck10MHz	44408	63793	96901	9.102	88.09	770.6	867.8
S4_M2_ck20MHz	44408	63801	46901	18.2	176.2	770.7	965.0
S4_M2_ck50MHz	44441	63321	17090	45.51	440.4	781.2	1267
S4_M2_ck100MHz	44441	63318	7090	91.02	880.9	781.2	1753
S4_M2_ck200MHz	44441	63319	2090	182.0	1940	781.2	2903
S4_M2_ck500MHz	45339	66186	0.03	456.6	4407	806.0	5669
S4_M4_ck10MHz	88730	127364	96898	18.21	176.2	1540	1735
S4_M4_ck20MHz	88730	127381	46898	36.41	352.4	1540	1929
S4_M4_ck50MHz	88793	126419	17090	91.02	880.9	1562	2533
S4_M4_ck100MHz	88793	126414	7090	182.0	1762	1562	3505
S4_M4_ck200MHz	88794	126417	2090	364.1	3879	1562	5805
S4_M4_ck500MHz	90873	131361	0.4	912.8	8814	1619	11300
S4_M8_ck10MHz	177397	254762	96897	36.4	352.3	3080	3469
S4_M8_ck20MHz	177397	254796	46897	72.81	704.7	3080	3858
S4_M8_ck50MHz	177516	253005	17090	182.0	1762	3123	5066
S4_M8_ck100MHz	177515	253003	7085	364.2	3524	3123	7011
S4_M8_ck200MHz	177515	253003	2086	728.2	7759	3123	11600
S4_M8_ck500MHz	180497	257564	0.2	1826	17600	3210	22600
S4_M16_ck10MHz	354713	509288	96897	72.49	704.4	6161	6938
S4_M16_ck20MHz	354713	509351	46897	145.0	1409	6161	7715
S4_M16_ck50MHz	354954	505759	17078	362.4	3522	6246	10100

Configuration	Design Area µM^2	Leaf Cell Count	Critical Path Slack ps	Net Switching Power µW	Cell Internal Power µW	Cell Leakage Power µW	Total Power µW
S4_M16_ck100MHz	354955	505759	7089	724.9	7044	6246	14000
S4_M16_ck200MHz	354954	505755	2089	1450	15500	6246	23200
S4_M16_ck500MHz	360931	514672	0.01	3635	35200	6418	45200
S4_M32_ck10MHz	709473	1019027	96897	145.6	1409	12300	13900
S4_M32_ck20MHz	709472	1019147	46900	291.3	2819	12300	15400
S4_M32_ck50MHz	709476	1019175	16897	728.2	7047	12300	20100
S4_M32_ck100MHz	709477	1019180	6897	1456	14100	12300	27900
S4_M32_ck200MHz	709476	1019172	1897	2913	31000	12300	46300
S4_M32_ck500MHz	724183	1039650	0.02	7306	70400	12900	90600
S4_M64_ck10MHz	1419321	2039762	96976	291.3	2819	24700	27800
S4_M64_ck20MHz	1419319	2039938	46978	582.5	5637	24700	30900
S4_M64_ck50MHz	1419322	2039991	16978	1456	14100	24700	40200
S4_M64_ck100MHz	1419320	2039993	6978	2913	28200	24700	55800
S4_M64_ck200MHz	1419318	2039980	1978	5825	62100	24700	92500
S4_M64_ck500MHz	1447122	2094657	0.0	14600	140800	25700	181200
S8_M1_ck10MHz	44079	63276	96446	9.102	87.89	764.6	861.6
S8_M1_ck20MHz	44079	63289	46499	18.21	175.8	764.6	958.6
S8_M1_ck50MHz	44114	62784	16592	45.51	439.4	775.2	1260
S8_M1_ck100MHz	44113	62781	6586	91.02	878.9	775.2	1745
S8_M1_ck200MHz	44113	62780	1586	182.0	1935	775.2	2892
S8_M1_ck500MHz	45119	63993	0.07	456.8	4398	802.6	5658
S8_M2_ck10MHz	88077	126243	96498	18.21	175.8	1528	1722
S8_M2_ck20MHz	88077	126265	46489	36.41	351.6	1529	1917
S8_M2_ck50MHz	88146	125257	16598	91.03	878.9	1550	2520
S8_M2_ck100MHz	88146	125258	6586	182.1	1758	1550	3490
S8_M2_ck200MHz	88145	125257	1586	364.1	3870	1550	5784
S8_M2_ck500MHz	90062	127696	0.03	913.6	8795	1601	11300
S8_M4_ck10MHz	176096	252487	96498	36.41	351.6	3056	3444
S8_M4_ck20MHz	176096	252521	46447	72.83	703.2	3056	3832
S8_M4_ck50MHz	176225	250611	16586	182.1	1758	3099	5039
S8_M4_ck100MHz	176225	250610	6586	364.1	3516	3099	6979
S8_M4_ck200MHz	176225	250613	1586	728.2	7739	3099	11600
S8_M4_ck500MHz	179044	250671	0.01	1827	17600	3185	22600

Configuration	Design Area µM^2	Leaf Cell Count	Critical Path Slack ps	Net Switching Power µW	Cell Internal Power µW	Cell Leakage Power µW	Total Power µW
S8_M8_ck10MHz	352107	505109	96485	72.83	703.1	6110	6886
S8_M8_ck20MHz	352108	505177	46445	145.7	1406	6110	7662
S8_M8_ck50MHz	352109	505174	16446	364.2	3516	6110	9990
S8_M8_ck100MHz	352109	505183	6447	728.3	7031	6110	13900
S8_M8_ck200MHz	352108	505178	1446	1457	15500	6110	23000
S8_M8_ck500MHz	357999	502229	0.0	3658	35100	6368	45200
S8_M16_ck10MHz	704193	1009962	96444	144.9	1406	12200	13800
S8_M16_ck20MHz	704193	1010095	46446	289.9	2812	12200	15300
S8_M16_ck50MHz	704195	1010123	16413	724.7	7030	12200	20000
S8_M16_ck100MHz	704195	1010127	6410	1449	14100	12200	27700
S8_M16_ck200MHz	704194	1010121	1411	2899	31000	12200	46100
S8_M16_ck500MHz	716736	1003013	0.0	7281	70300	12700	90300
S8_M32_ck10MHz	1408749	2021994	96412	291.3	2812	24400	27600
S8_M32_ck20MHz	1408770	2022264	46415	582.6	5625	24400	30700
S8_M32_ck50MHz	1408841	2022289	16414	1456	14100	24400	40000
S8_M32_ck100MHz	1408845	2022295	6415	2913	28100	24500	55500
S8_M32_ck200MHz	1408838	2022276	1414	5826	61900	24500	92200
S8_M32_ck500MHz	1436376	2025698	0.0	14600	140600	25500	180700
S16_M1_ck10MHz	87752	125586	96364	18.21	175.6	1522	1716
S16_M1_ck20MHz	87752	125609	46366	36.41	351.2	1522	1910
S16_M1_ck50MHz	87752	125609	16366	91.03	878.0	1522	2491
S16_M1_ck100MHz	87751	125606	6362	182.4	1757	1522	3462
S16_M1_ck200MHz	87750	125599	823.27	364.1	3865	1522	5751
S16_M1_ck500MHz	89940	130282	0.06	912.9	8784	1604	11300
S16_M2_ck10MHz	175443	251322	96364	36.41	351.2	3043	3430
S16_M2_ck20MHz	175441	251361	46366	72.82	702.3	3043	3818
S16_M2_ck50MHz	175440	251364	16195	182.1	1756	3043	4981
S16_M2_ck100MHz	175442	251370	6365	364.1	3512	3043	6919
S16_M2_ck200MHz	175438	251355	818.82	728.1	7729	3043	11500
S16_M2_ck500MHz	179061	250014	0.07	1825	17500	3192	22600
S16_M4_ck10MHz	350831	502345	96193	72.81	702.3	6086	6861
S16_M4_ck20MHz	350825	502404	46365	145.6	1405	6086	7636
S16_M4_ck50MHz	350824	502412	16250	364.0	3512	6086	9961

Configuration	Design Area μM^2	Leaf Cell Count	Critical Path Slack ps	Net Switching Power µW	Cell Internal Power µW	Cell Leakage Power µW	Total Power µW
S16_M4_ck100MHz	350824	502409	5990	728.1	7023	6086	13800
S16_M4_ck200MHz	350819	502384	820.6	1456	15500	6085	23000
S16_M4_ck500MHz	358063	499503	0.01	3652	35100	6380	45100
S16_M8_ck10MHz	701624	1004766	95980	145.6	1405	12200	13700
S16_M8_ck20MHz	701613	1004884	46365	291.2	2809	12200	15300
S16_M8_ck50MHz	701615	1004919	16365	728.1	7023	12200	19900
S16_M8_ck100MHz	701616	1004926	6194	1456	14000	12200	27700
S16_M8_ck200MHz	701601	1004855	818.67	2912	30900	12200	46000
S16_M8_ck500MHz	717298	1013692	0.01	7303	70200	12700	90200
S16_M16_ck10MHz	1403530	2010567	95956	291.2	2809	24400	27500
S16_M16_ck20MHz	1403542	2010837	45617	582.5	5618	24400	30600
S16_M16_ck50MHz	1403571	2010856	16215	1456	14000	24400	39900
S16_M16_ck100MHz	1403573	2010842	5959	2912	28100	24400	55400
S16_M16_ck200MHz	1403566	2010810	787.02	5825	61800	24400	92000
S16_M16_ck500MHz	1435709	2036522	0.06	14600	140300	25600	180500
S32_M1_ck10MHz	175119	250777	95172	36.34	351.0	3036	3423
S32_M1_ck20MHz	175116	250807	45181	72.68	701.9	3036	3811
S32_M1_ck50MHz	175451	249117	15221	181.9	1755	3024	4961
S32_M1_ck100MHz	175453	249142	5202	363.9	3510	3024	6898
S32_M1_ck200MHz	175450	249123	179.62	727.8	7726	3024	11500
S32_M1_ck500MHz	178245	245043	0.25	1822	17500	3166	22500
S32_M2_ck10MHz	350186	501238	95173	72.7	702.0	6073	6847
S32_M2_ck20MHz	350182	501299	45180	145.4	1404	6073	7622
S32_M2_ck50MHz	350992	495849	15348	363.9	3510	6133	10000
S32_M2_ck100MHz	350991	495839	5343	727.8	7020	6133	13900
S32_M2_ck200MHz	350858	497901	176.2	1455	15500	6049	23000
S32_M2_ck500MHz	356949	491874	0.02	3645	35100	6343	45100
S32_M4_ck10MHz	700327	1002556	94331	145.4	1404	12100	13700
S32_M4_ck20MHz	700319	1002698	45166	290.8	2808	12100	15200
S32_M4_ck50MHz	701661	995774	15175	727.9	7020	12100	19800
S32_M4_ck100MHz	701651	995714	5151	1456	14000	12100	27600
S32_M4_ck200MHz	701652	995727	162.58	2912	30900	12100	45900
S32_M4_ck500MHz	718792	1009055	0.0	7296	70200	12800	90200

Configuration	Design Area µM²	Leaf Cell Count	Critical Path Slack ps	Net Switching Power µW	Cell Internal Power µW	Cell Leakage Power µW	Total Power µW
S32_M8_ck10MHz	1400976	2006171	95105	290.8	2808	24300	27400
S32_M8_ck20MHz	1400978	2006454	45139	581.7	5615	24300	30500
S32_M8_ck50MHz	1403573	1992546	15133	1456	14000	24200	39700
S32_M8_ck100MHz	1403566	1992532	5122	2912	28100	24200	55200
S32_M8_ck200MHz	1403559	1992521	113.09	5824	61800	24200	91800
S32_M8_ck500MHz	1437566	2010748	0.01	14600	140300	25600	180500
S64_M1_ck10MHz	349880	500714	94344	72.58	701.7	6064	6839
S64_M1_ck20MHz	349874	500798	45179	145.2	1404	6064	7613
S64_M1_ck50MHz	350536	497494	14010	363.4	3509	6040	9912
S64_M1_ck100MHz	350536	497492	4159	726.8	7018	6040	13800
S64_M1_ck200MHz	350660	495576	49.01	1453	15500	6041	22900
S64_M1_ck500MHz	356076	488539	0.04	3643	35100	6318	45000
S64_M2_ck10MHz	699734	1001685	94342	145.2	1403	12100	13700
S64_M2_ck20MHz	699719	1001808	45243	290.4	2807	12100	15200
S64_M2_ck50MHz	701011	994947	14121	726.7	7017	12100	19800
S64_M2_ck100MHz	701020	995010	4239	1453	14000	12100	27600
S64_M2_ck200MHz	701277	991148	44.58	2907	30900	12100	45900
S64_M2_ck500MHz	717915	1001009	0.67	7286	70100	12800	90200
S64_M4_ck10MHz	1399763	2004253	94237	290.4	2807	24300	27400
S64_M4_ck20MHz	1399761	2004507	44019	580.7	5614	24300	30500
S64_M4_ck50MHz	1402229	1990703	14050	1454	14000	24200	39700
S64_M4_ck100MHz	1402242	1990767	4880	2907	28100	24200	55100
S64_M4_ck200MHz	1402853	1982215	7.86	5814	61800	24200	91800
S64_M4_ck500MHz	1436402	2003027	0.0	14600	140300	25500	180400
S128_M1_ck10MHz	699468	1001266	93462	145.0	1403	12100	13700
S128_M1_ck20MHz	699441	1001351	43322	290.1	2807	12100	15200
S128_M1_ck50MHz	700716	994472	13809	726.1	7017	12100	19800
S128_M1_ck100MHz	700960	991057	4033	1452	14000	12100	27600
S128_M1_ck200MHz	701046	992147	0.0	2905	30900	12100	45900
S128_M1_ck500MHz	717477	997898	0.0	7278	70100	12700	90100
S128_M2_ck10MHz	1399207	2003368	93133	290.1	2807	24300	27400
S128_M2_ck20MHz	1399214	2003633	42824	580.2	5613	24300	30500
S128_M2_ck50MHz	1401655	1990000	13533	1452	14000	24200	39600

Configuration	Design Area µM^2	Leaf Cell Count	Critical Path Slack ps	Net Switching Power µW	Cell Internal Power µW	Cell Leakage Power µW	Total Power µW
S128_M2_ck100MHz	1402201	1981812	3815	2905	28100	24200	55100
S128_M2_ck200MHz	1402541	1983804	0.0	5809	61800	24200	91800
S128_M2_ck500MHz	1434986	1998524	-0.06	14600	140200	25500	180300
S256_M1_ck10MHz	1399401	1954970	92768	290.0	2807	24100	27200
S256_M1_ck20MHz	1399391	1955154	41094	580.0	5613	24100	30300
S256_M1_ck50MHz	1401664	1946639	11939	1450	14000	24100	39600
S256_M1_ck100MHz	1403276	1935280	4024	2900	28100	24200	55100
S256_M1_ck200MHz	1403114	1934632	44.52	5800	61800	24200	91700
S256_M1_ck500MHz	1432585	1992355	0.0	14600	140200	25400	180200