McGill
Electrical &
Computer
Engineering

304-487
Computer Architecture Laboratory

Project Compendium

2009/2010

Professor Cooperstock
Department of Electrical and Computer Engineering

A Hardware Fourier Tag Decoder

Anqi Xu and François Côté
{anqi.xu, francois.cote}@mail.mcgill.ca

Abstract **– Fourier Tag is a barcode-like fiducial marker system, designed primarily as a vision-based input interface for a robot controller [1]. The numerical ID of each marker is embedded in the frequency spectrum of the tag's grayscale pattern, and can be interpreted by the robot.**

This project involves the design of a Fourier Tag decoder using FPGA technology. In the context of the robot, such dedicated hardware would remove the need of a software decoder, which would otherwise be competing with the robot controller for precious CPU time. As a consequence, the controller's sampling time would be reduced, which would result in a more stable and responsive robot.

The implementation consists of the following algorithmic tasks: sampling the input image, averaging the samples, transforming them using a pre-built FFT core [2], thresholding the resulting magnitude spectrum, and finally decoding the numerical ID. The main performance criterion involves surpassing the decoding speed of a reference marker software (ARTag) [3] that is currently used to control robots.

Index Terms **– Fourier Tag, fiducial marker, human-robotics interaction, FPGA, VHDL**

I. INTRODUCTION

The use of visual markers as a close-range input interface for robots is a relatively new and unexplored subject. Its main advantages, such as low cost, portability, and robustness to noise, can only be truly appreciated when employed in extreme environments, such as underwater, where conventional communication methods would be either impractical or plainly impossible to implement.

Dudek et al. explored the possibility of using symbolic tags to control an underwater robot named AQUA [1], by mapping the detected marker IDs into a sequence of robot instructions and executing them on-the-fly. In their first implementation, they used a marker system called ARTag [3] (see Figure 1.a for a sample ARTag marker). These visual symbols are extremely resilient to image distortions, due to the vast abundance of redundancy in the system, under the form of error correction codes. Unfortunately, a substantial portion of the image is required to represent these codes; therefore the size of useful information that can be embedded is severely restricted.

This paper introduces a new fiducial marker system called Fourier Tag, through the description of a particular implementation of its decoder in hardware. The decoder project is aimed particularly at the Human-Robotics Interaction application domain, as a potential replacement for ARTag markers in the context of the AQUA robot. The Fourier Tag protocol is designed to feature similar robustness qualities to those of the ARTag system, although the crucial difference is that this new protocol aims to reduce the space occupied by redundant information, while still preserving roughly the same amount of redundancy. Such a seemingly oxymoronic design criterion is made possible by encoding the binary data in the frequency spectrum of the marker image.

(a) (b)

Fig. 1. a) An ARTag marker encodes data using black and white squares; b) A Fourier Tag marker embeds data into the frequency spectrum of the image.

A Discrete Fourier Transform algorithm is required to decode Fourier Tags. Because the hefty computational strain imposed by software DFT algorithms conflict with the allocatable amount of CPU time for the robot controller, this project proposes to move the decoder onto an FPGA board, thus allowing the controller to have more frequent access to the CPU, and therefore making the robot more stable and responsive.

II. THE FOURIER TAG PROTOCOL

A. Generator Algorithm

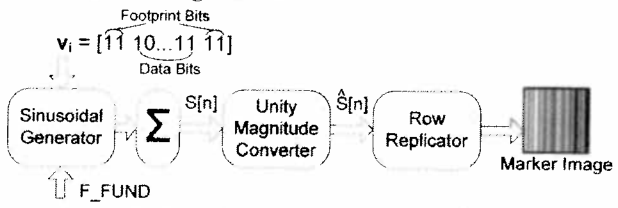


Fig. 2. The Fourier Tag generator algorithmic flow diagram

The Fourier Tag protocol embeds a binary data vector $\mathbf{v}_i$ of length I in the frequency magnitude spectrum of a generated image. In particular, the periodic marker sequence S[n] is composed of sinusoids whose frequencies are integer multiples of a pre-determined fundamental frequency, F_FUND.

$$S[n]=\sum_{i=1}^{I} v_i \cdot \sin\left[2\pi\left(i\cdot F\_FUND\right)\cdot n\right];$$

The binary vector $\mathbf{v}_i$ contains a small footprint pattern along with the rest of the data. The footprint bits, located on both sides of the data bits in two equal parts, are asserted in every marker and serves as a means to identify the Fourier Tag. This positioning ensures that if some high frequency contents were not detected, then the image would be dismissed due to the absence of a full footprint pattern, rather than having the data bits being truncated and possibly misinterpreted.

The marker image is created from the summed sequence S[n] by scaling the magnitudes to unity, duplicating the horizontal sequence vertically, and then interpreting the magnitudes as grayscale intensities.

B. Decoder Algorithm

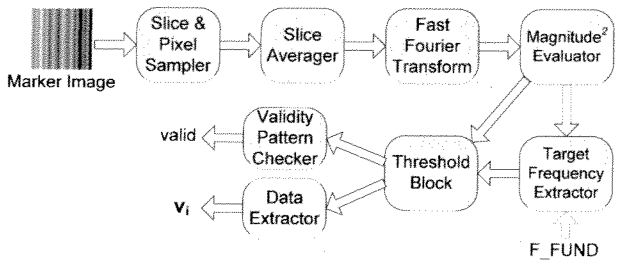


Fig. 3. The Fourier Tag decoder algorithmic flow diagram.

The decoder aims to recover binary data from the frequency magnitude spectrum of a supplied image, which may contain some noise. Given a 2-dimensional array of grayscale values (i.e. the supplied source image), several horizontal rows (hereby referred to as *slices*) are chosen at fixed intervals. These slices are then averaged to attenuate the potential additive noise in the image, possibly due to non-uniform lighting or partial occlusion effects. Under these non-ideal conditions, the averaged sequence will still be relatively similar to the ideal sequence, assuming that the distortions are not overwhelming.

A Fast Fourier Transform (FFT) algorithm is used to obtain the frequency spectrum of the averaged slice. The Fast Fourier Transform is a family of efficient algorithms that computes the Discrete Fourier Transform of a sequence. The output of the FFT consists of two sequences, representing the real and imaginary frequency values, up to a maximum frequency.

This particular Fourier Tag implementation only embeds information in the *magnitude* spectrum of the image (as opposed to the *phase* spectrum). To minimize resources, the *power* spectrum (i.e. the *squared magnitude* spectrum) is evaluated, by summing the squares of both the real and imaginary values.

Next, the magnitudes at particular frequencies (i.e. at multiples of the fundamental frequency) are extracted from the complete data set. These values are then converted into binary format, by thresholding the squared magnitudes with one fourth of the maximum value (i.e. half of the maximum magnitude). The resulting bit vector contains both the data bits and the footprint bits, which can then be easily extracted and validated, respectively.

At this point, a word of caution is due – the algorithms described above are geared towards the simplified version of the general Fourier Tag protocol discussed in this paper, and thus by no means represent a complete documentation for generating and decoding markers using the full Fourier Tag protocol.

III. DESIGN METHODOLOGY

A top-down design methodology was taken to separate the decoder algorithm into different subsections, which could then be implemented as individual hardware modules. Because the heart of the decoder algorithm involved computing the FFT, the design was split into three stages – a central core consisting of the FFT, sandwiched between an image-processing pre-FFT stage, and a data analysis post-FFT stage.

Because the ultimate evaluation criterion for this FPGA-based design is its processing speed, the general design approach was established to build a system using as much combinational circuitry as possible, which would minimize the system's latency by sacrificing resources. Unfortunately, not all the steps in the algorithm could be realized using combinational logic – the FFT and the maximum magnitude detector blocks are both sequential in nature. Therefore, the design procedure involved building the sequential core first, and then appending onto it the pre- and post-FFT combinational sections.

Since the Fourier Tag protocol and the decoder algorithm are first introduced in this paper, it means that no previous implementations exist for this hardware decoder project to rely on. To ensure that the algorithm and the hardware design were correct, a software Fourier Tag decoder function was coded using MATLAB®. This function was written in such a way to imitate the hardware design as much as possible, so that all logical flaws associated with this particular design would be fixed prior to implementing it in VHDL!

Even with an emulated function written in MATLAB®, it was not guaranteed that the hardware implementation would succeed, due to complications that could arise only in the hardware domain, such as lack of data precision and race conditions. To ensure that the hardware implementation would have a high chance of success, three design goals were set in place – the hardware design had to be *modular, parametric, and scalable.* The implementation assigned variables to all parameters that could potentially affect the outcome of the system, such as the data resolution, the number of slices used, and the number of samples used. Additionally, each step in the decoder algorithm was broken down into separate, self-containing modules, so that these modules could be tested for correctness independently. Essentially, the hardware system was designed for test.

To reduce on development time, it was decided that this hardware implementation would use a pre-built FFT core. After considering many implementations, including those from OpenCores.ORG and from Xilinx®, the Altera® FFT MegaCore function [2] was chosen, for its parametric and scalable design, and for its extensive documentation.

As a consequence from choosing the Altera® FFT core, the target device was constrained to be a member of one of Altera®'s FPGA families. The Stratix II device family was chosen for its abundance of resources, and its extremely swift circuitry. This decision perfectly reflects the design approach of trading resources (including cost) for speed.

Finally, two critical assumptions were made during the design of the decoder system – the decoder assumes that an external Fourier Tag *detector* delivers to it potential marker regions *of uniform dimension*, extracted from a source image. Furthermore, it assumes that these regions contains a fixed number of periods of the data sequence (H_PRD), so it would be the *detector*'s job to resize the regions before sending them to the decoder.

IV. IMPLEMENTATION

A. Pre-FFT Stage

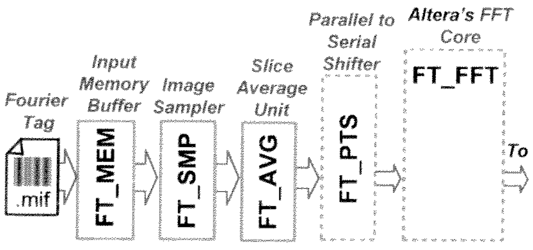


Fig. 4. Block diagram for the pre-FFT stage

The pre-FFT stage focuses on converting a 2-dimensional region into a 1-dimensional sequence of grayscale values, ready to be processed by the FFT core.

The first module in the pre-FFT stage is the memory unit (FT_MEM.vhd). Being designed for simulation purposes only, its function consists of reading in the Fourier Tag region from a flat file, in binary format. Activated by a reset signal, the memory stores the marker region and allows it to be accessed by the rest of the hardware in its entirety. In reality, this module simply represents the wirings between the Fourier Tag *detector* and the decoder. Directed into the sampling module (FT_SMP.vhd), the data lines holding the stored bits are selectively reconfigured. With the knowledge of the fixed image dimensions, this module samples the data into 2^{SLICE_EXP} slices and 2^{SAMPLE_EXP} pixels, with the chosen slices and pixels taken at constant intervals from the original array.

The previous module introduces a resource-saving optimization. The sampling theorem [4] dictates that a sequence only needs to be sampled slightly more than twice per its smallest period, for the *sampled* sequence to have identical frequency contents as the original one. The pre-FFT stage thus down-samples every slice to reduce the length of the sequence, which results in a smaller FFT core. This optimization is only possible because the fundamental frequency is assumed to be fixed and known.

Because the tag information is repeated vertically, ideally only one slice of the image is necessary to fully describe it. However, in order to promote robustness, the hardware requires the average of several sampled slices. The averaging block (FT_AVG.vhd) performs the addition of the slice elements (i.e. the sampled pixels) in parallel using a divide-and-conquer topology. The sums are then right-shifted through rewiring, as an efficient way to divide by the total slice count. Although this approach is resource efficient and scalable, it requires the number of slices and samples to be powers of 2.

Since the FFT core is sequential, a standard parallel-to-serial block shift register (FT_PTS.vhd) is required to convert the data set into serial form, for further processing.

B. FFT Stage

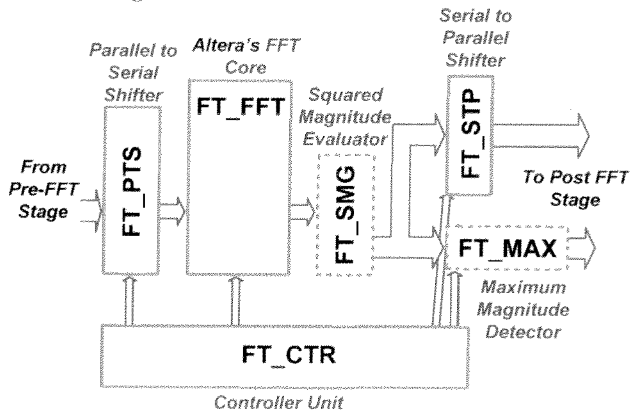

From Pre-FFT Stage
Parallel to Serial Shifter
FT_PTS
Altera's FFT Core
FT_FFT
Squared Magnitude Evaluator
FT_SMG
Serial to Parallel Shifter
FT_STP
To Post FFT Stage
FT_MAX
Maximum Magnitude Detector
FT_CTR
Controller Unit

Fig. 5. Block diagram for the FFT stage

The Altera® FFT core operates according to a point-to-point interface protocol named *Atlantic*. Each element inside the averaged slice is streamed to the core at every clock cycle. This event is initiated using a hand-shaking protocol – the central controller first asserts two 'data-available' control signals (sink_dav, source_dav), and once the core is ready, it responds by asserting a 'ready-to-receive' signal (sink_en). Once the initial communication has been established, the central controller then shifts the data set into the core, element by element, while simultaneously asserting a 'start-of-packet' pulse (sink_sop). Once the transform is ready to present the first output element, it asserts its *own* 'start-of-packet' pulse (source_sop) and simultaneously shifts out the results. This core expects data inputs and data outputs on pairs of ports – one for the real part and one for the imaginary part. In this particular case, the imaginary input port is never used.

For this application, the core (FT_FFT.vhd) is generated to have 256 data points and 8 bits for data precision, using the streaming mode. The multiplication operation is realized by using a 3-multipliers-5-adders structure. This fixed parameter setup adds a slight complexity to the overall parametric hardware design – when the pre-FFT sequence length is changed to a smaller value (by decreasing the sampling rate, or equivalently SAMPLE_EXP) than the number of FFT data points, the core automatically pads '0's to the end of the smaller sequence. Thus, the FFT implicitly up-samples of the input slice, and even though this change does not remove any information, the post-FFT stage must compensate for this effect.

Because the FFT core uses signed arithmetic operations, it is wasteful to represent the input grayscale values using an unsigned format. Thus, the binary dump of the marker image is generated (using a simple MATLAB® function) using two's complement representation.

C. *Post-FFT Stage*

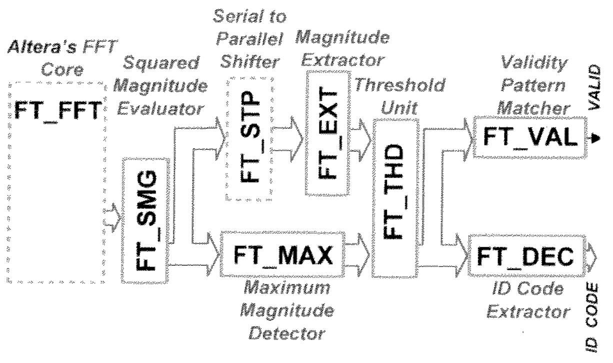


Fig. 6. Block diagram for the post-FFT stage

The post-FFT stage may appear overwhelming at first, due to the numerous modules encompassed by it. However, the actual algorithmic goal is fairly simple – look at the power spectrum at particular frequencies, and identify peaks as binary '1's and the lack of peaks as '0's.

The squared magnitude evaluator module (FT_SMG.vhd) is placed at the output of the FFT core, so that the real and imaginary frequency values can be immediately transformed into magnitudes, thus halving the data set and saving on resources. An additional advantage for evaluating the magnitudes before converting the data set back into parallel form, is that only one module is required for the entire data set, because the set is processed by this module sequentially.

Since most of the post-FFT modules can be expressed using combinational logic, the output data set needs to be converted back into parallel form before it can be analyzed. A standard serial-to-parallel block shift register (FT_STP.vhd) is employed to achieve this effect. However, to fully exploit the sequential nature of the FFT stage, the sole remaining sequential module – the maximum value detector (FT_MAX.vhd) – is set to execute in unison with the shift register. This module evaluates and stores the maximum data among all of its previously seen inputs by using a comparator (FT_CMP.vhd) and an internal register. Since the data precision is parametrically defined, the greater-than-or-equals comparator is coded using a recursive algorithm, based on a 1-bit comparator.

After the data set has been converted back into its parallel form, a frequency extractor (FT_EXT.vhd) module selects the magnitudes at multiples of the fundamental frequency F_FUND. Because this implementation assumes 8 data bits and 4 signature bits, the extractor module returns a vector of length 12. It is worth noting that the length of the input sequence is no longer $2^{\text{SAMPLE\_EXP}}$, but rather N (i.e. the number of FFT data points), since the FFT core implicitly up-samples the input sequence (of length $2^{\text{SAMPLE\_EXP}}$) to its own length N, as mentioned previously.

$$V[\mathrm{i}]=\mathrm{D}\left[(\alpha-1)+(\alpha)(\mathrm{i})\left(\frac{\mathrm{F\_FUND}}{\dfrac{\mathrm{N}}{2\bullet\mathrm{H\_PRD}}-1}\right)(\mathrm{N})\right]=$$
$$=\mathrm{D}\left[(\alpha-1)+(\alpha)(\mathrm{i})\left(\mathrm{F\_FUND}\bullet\frac{\mathrm{N}\bullet\mathrm{H\_PRD}}{\mathrm{N}-2\bullet\mathrm{H\_PRD}}\right)\right];$$
$$\alpha=2^{\mathrm{FFT\_N\_EXP}-\mathrm{SAMPLE\_EXP}};$$

The index equation shown above determines the locations of the subset elements V[i] (of length I = 12) from the input sequence D[i] (of length N = 256) by first scaling the discrete frequency range by the ratio of the fundamental frequency F_FUND over the maximum frequency N/(2*H_PRD) – 1. Recall that H_PRD, the number of periods in the input region, is assumed to be fixed and known beforehand. A second scaling factor, N, is introduced to transform the range from [0, 1] into the integer version [0, N]. Without considering the DC component ($\omega = 0$ rad/s), which will be discussed in the controller section, the desired frequency values are simply the first I elements, at least in theory. Unfortunately, due to the implicit up-sampling of the FFT core, the frequency spectrum is divided by $\alpha = N - 2^{\mathrm{SAMPLE\_EXP}} = 2^{\mathrm{FFT\_N\_EXP - SAMPLE\_EXP}}$. This means that there are $(\alpha - 1)$ zeros between each desired element, so the offset $(\alpha - 1)$ and the index factor α are added to compensate for the up-sampling effect.

The threshold module (FT_THD.vhd) employs the same greater-than-or-equals comparator used previously, to compare all the elements in the extracted data set to a threshold value. This threshold is naturally chosen as the maximum magnitude over 2, which, in the power spectrum, is equivalent to the maximum value right-shifted by 2 bits (i.e. divided by 4). The bit vector output of the threshold module is simply the outputs of the individual comparators, inside this block.

At this point, the target frequency values are now in binary format. The remaining two tasks consist of extracting the data bits through simple rewiring (FT_DEC.vhd), and validating the footprint bits. The validity pattern checker (FT_VAL.vhd) filters the raw bit vector with a mask, to isolate only the bits that are relevant to the common Fourier Tag footprint. All the bits of the masked vector are AND'ed with each other using a special sub-module (FT_VectorGate.vhd). This sub-module is recursively coded using a divide-and-conquer algorithm, to accommodate for vectors of arbitrary length.

D. *Controller*

The central controller (FT_CTR.vhd) is required to guarantee synchronism between the sequential components. This block consists of an 8 state Finite State Machine, which initiates all FFT core transactions. The controller governs the data flow by invoking the pre-FFT and post-FFT shift registers (inshift, outshift).

In addition to maintaining synchronism, the controller can operate selectively with the data transfers. Due to excessive presence of noise in the first pass through the core, the controller only asserts completion of the transform once the second pass is ready (ready). Another data anomaly is the zero frequency impulse – the first element to emerge from the core corresponds to the presence of a DC component. The presence of this offset is traced to sinusoidal additions in generating the Fourier Tag. Without omitting this impulse, other frequency components appear relatively small and can pass undetected. Luckily, due to its entirely predictable position, the controller may block the artefact from later parts of the hardware.

The states of the controller are detailed below. Figure 7 illustrates the transition flow of the states.

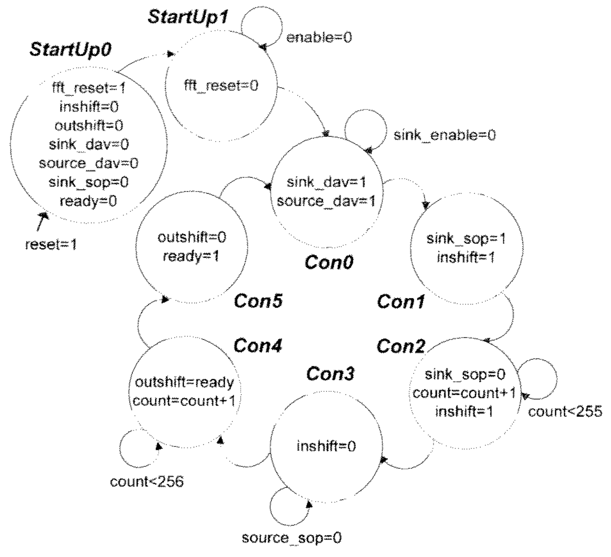


Fig. 7. State diagram for the controller using a 256 point FFT core

STATE	DESCRIPTION
StartUp0	All control signals are set to 0 except for the FFT reset signal.
StartUp1	The FFT reset signal is unasserted; the core is reset.
Con0	The FFT core is advised that data is available.
Con1	When the core acknowledges the availability of the data, it is advised of the 'start-of-packet' signal. The parallel-to-serial shift register is enabled.
Con2	The 'start-of-packet' streaming signal is unasserted.
Con3	After shifting in 256 data elements, the parallel-to serial shift register is disabled.
Con4	Once the FFT core asserts its own 'start-of-packet' signal for the output data streaming, *and if this is the second pass of the data through the core*, the serial-to-parallel register is enabled.
Con5	After shifting out 256 transformed elements, the serial-to-parallel shift register is disabled.

Table 1: State descriptions for the controller's Finite State Machine, using a 256 point FFT core

E. *Testbench*

The testing of the Fourier Tag decoder uses several testbench versions. In this section, the final testing solution is described (see section V for the complete testing methodology).

The testbench (FT_TSB.vhd) is designed to feed the hardware system with known tags and verify the output.

Making use of MATLAB®, the binary representation for *all possible* tag inputs are pre-generated into flat files. During testing, the memory unit is instructed to read in the tags one after another. With an inscribed code of 8 bits, 256 tag files are accessed by the memory unit. To enhance the control over testing, the tags are fed in order, from 0 to 255. As a consequence, the testbench uses a counter to keep track of what the decoded tag should be.

In due course, the testbench verifies if the output corresponds to the tag code, or more precisely, to the count value. If the decoded value does not match the count, a report statement flags the error during simulation.

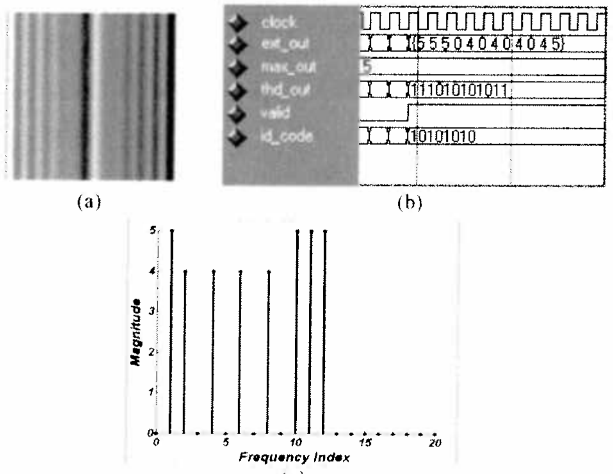
Once the decoder passes all tags, another version of the testbench is used to determine the processing time. Here, VHDL report statements issue a timestamp at the beginning and at the end of the decoding process.

V. EVALUATION AND OPTIMIZATION

A. Correctness Evaluation

The first step in testing consists of evaluating the functionality of each of the individual system blocks. The design modules are put together, block by block, and the gradually built-up system is tested for correctness before each additional block is added.

Because many combinational modules have a hardware topology that directly reflects their function (for example, rewiring), the RTL schematics of the synthesized blocks are used as a visual indication of correctness. Once visual correctness is ascertained, *random* test cases are chosen to verify the functionality. As the system is being integrated as a whole, the testbench is modified accordingly to make the output of the last block accessible via a flat file. The data can then be fed into MATLAB®, for verification.



(c)

Fig. 8. a) A Fourier Tag with the inscribed code [0 1 0 1 0 1 0 1];
b) The corresponding waveform for the output of the decoder: the tag is shown decoded (little-endian encoding) with the validity asserted;
c) The MATLAB® plot of the FFT core's output.

At this preliminary stage of testing, only the *apparent corner cases* (such as tags with all 1's, 0's, or alternating patterns) are fully verified. This approach leads to an early calibration of the system. Specifically, it gives an indication of the required threshold to properly convert the frequency content to a binary form.

B. Accuracy Evaluation & Optimization

Accuracy evaluation depends directly on the parameters to which the decoder is set. These main system specifications include: the tag pixel resolution (RES), the number of samples ($2^{\text{SAMPLE\_EXP}}$) and the number of slices ($2^{\text{SLICE\_EXP}}$).

The resolution corresponds to the amount of shades of gray possible used to represent the frequency content. In its binary format, each pixel can take on RES different values. The resolution describes the precision with which one can represent the oscillations of the grayscale tag. The lower the resolution, the greater the quantization noise, and the spectral peaks are less pronounced. The number of samples relate to how many pixels per horizontal row are chosen to represent the frequencies of the system.

There are a total of $2^{\text{SAMPLE\_EXP}}$ linearly spaced sampled pixels. Choosing too little samples may lead to aliasing. The number of slices relate to how many horizontal rows of pixels are chosen.

There are a total of $2^{\text{SLICE\_EXP}}$ linearly spaced slices taken from the original tag image. The number of slices is a direct measure of robustness. As explained previously, when faced with partially corrupted tags, more slices means more chances to overcome the distortion. Using large values for these specifications undoubtedly leads to successful decoding. Unfortunately, this comes at the price of unnecessary resource usage.

The software (MATLAB®) version of the decoder proved to be dependant on the system specifications to perform adequately. Varying these specifications caused the system to fail while decoding certain tags. Therefore, at the hardware level, similar results were expected. However, it must be clear that a system which cannot decode 100% of tags is considered be a failure and therefore is unacceptable!

Because half the system deals with frequency content, it becomes evident that the previously investigated *apparent corner cases* are not really corner cases for all the components of the design. Therefore, an exhaustive testing approach must be used. Conforming to the format of an inscribed 8 bit code, all 256 possible tags are generated and automatically fed into the decoder. In guise of an exhaustive testing strategy, it is impractical to show the waveforms resulting from all the tests. The system either passes or fails in decoding the tag. What is observed is repetitious of the waveform illustrated in Figure 8.b.

The strategy in determining the minimal specifications consists in fixing two parameters and varying the third. The minimum parameter which still leads to a successful run is selected and fixed.

The first order of business consists of determining the minimal number of samples before aliasing sets in. This occurs when taking 64 samples (SAMPLE_EXP = 6). Next, it is observed that the resolution fails at a 5 bit level (RES = 5). Here, the lack of precision causes noticeable frequency impulses to miss the threshold.

In order to test the required number of slices, an obstructed image is fed into the system. In this case all tested SLICE_EXP values succeed in extracting the tag. However, due to the nature of the slicing, it may occur that the slices are sampled completely around the obstruction. Here, it is more important that higher orders of slicing perform just as well; demonstrating the success of the averaging. Therefore a reasonable minimum value is set to 2, to insure coverage of the tag edges and center region. The minimal specifications are therefore: RES = 6, SLICE_EXP = 2, and SAMPLE_EXP = 7.

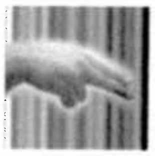


Fig. 9. A partially obstructed Fourier Tag image

C. Resource Utilization

A decision is made to add a small margin to each of the minimal specification parameter values. This safety net ensures the resiliency of the system when faced with unaccounted disturbances. To assess the cost of these margins, the code is synthesized using both the minimal specs and those with the margins.

	Minimal Specs RES = 6, SLICE_EXP = 2, SAMPLE_EXP = 7	**Minimal Specs + Margins** RES = 7, SLICE_EXP = 3, SAMPLE_EXP = 8
IO	34 / 343 (9.91 %)	36 / 343 (10.50 %)
LUT	3205 / 27104 (11.82 %)	14511 / 27104 (53.54 %)
REG	4121 / 29034 (14.19 %)	5673 / 29034 (19.54 %)
DSP	2 / 128 (1.56 %)	2 / 128 (1.56 %)
Clock	6.43 nsec (155.44 MHz)	6.56 nsec (152.35 MHz)

Table 2: Resource utilization data, after synthesizing the system using both set of specs, for the device: Altera Stratix II EP2S30F484 Grade C

Table 2 illustrates that the cost of using the margins is a four-fold increase of LUTs. This drastic gain can be explained by remembering that both SLICE_EXP and SAMPLE_EXP represent the 2's powers of the number of slices and samples. Therefore, if these two parameters are both incremented by 1, this corresponds to doubling the number of slices and the number of samples, thus explaining the four-fold increase of look-up tables needed to process the image in the pre-FFT stage.

Fortunately, the critical path for both set of specifications is found through the sequential controller, which is relatively undisturbed by changes in the parameters (other than possibly updating the total number of cycles it takes for the FFT core to process a slice). Because the speed of the system is not severely hampered by the margins, the set of specifications with the margins is designated as the final system configuration.

D. Latency Evaluation

The final and most important evaluation phase involves comparing the processing time for the finalized FPGA system to that of its benchmark – the ARTag software decoder function.

The duration it takes for the Fourier Tag decoder to process a marker is determined by putting VHDL report statements in the testbench (FT_TSB.vhd) before passing the input image and after receiving the validity flag. This simple approach makes the assumption that the marker has been properly detected. Otherwise, the validity flag would not be asserted.

To obtain the elapsed durations after processing multiple markers through the system, the report statement outputs are dumped into a flat file and are passed to a simple text parser application that calculates the time differences and also evaluates the average elapsed duration. Results from the parser show that all time differences are identical. This makes sense because the hardware propagation time is constant for all possible marker inputs.

Obtaining the processing time for the ARTag software decoder function is equivalent to finding the wall-clock execution time required by the function. The ARTag timing testbench is a small application written in C that generates all possible ARTag markers and then measures the execution time the decoder function took to process through each tag. These time differences are then tallied and averaged. In contrast to the Fourier Tag hardware system, the execution times actually improved as the decoder processed through more markers. This effect can be accredited to the marker buffer array finding itself into the system's cache due to its frequent usage. This buffered setup is employed in reality, since the robot constantly scans for tags because it does not know when to expect the next marker.

After determining the clock speed for the Fourier Tag decoder, a final version of the system is simulated while noting the timing information. For the purpose of evaluation, the ARTag timing testbench is executed on the hardware of the actual AQUA robot. In comparing the timing reports, the following results are obtained.

	Fourier Tag Decoder ModelSim Simulation	ARTag Decoder Timing Testbench Application
Sample Data Set	22008 nsec	8364 usec
	22008 nsec	10825 usec
	22008 nsec	7718 usec
	22008 nsec	9274 usec
Average	22 usec	9145 usec

Table 3: Sample data & average processing times for Fourier Tag Decoder ModelSim simulation and the ARTag Decoder timing testbench

VI. CONCLUSION

This paper proposed the hardware implementation of a decoder for a novel barcode-like system called Fourier Tags. The decoder samples the tag pixels, extracts the spectral content and reads the inscribed code. The hardware strategy of a scalable, parametric and modular design allowed for full control and observability. In turn, this allowed for pinpointing the minimal design specifications.

To achieve reliability, the minimal specifications were increased by a small margin. The final specifications describe taking 128 samples and 8 slices from an 8 bit resolution image. In the scope of the robotic application for which it is conceived, the Fourier Tag decoder has proven to be 415 times faster than then current technology.

VII. FUTURE WORK

There are many possibilities in terms of potential improvements and additions to the hardware Fourier Tag decoder, although perhaps the most interesting and valuable addition would be to fulfill the assumptions made by this decoder project – design and build a Fourier Tag hardware *detector*, that can determine potential marker-looking regions in a given image, and send the cropped regions to the *decoder* for analysis.

Other interesting work includes:

- Encoding data in the phase spectrum
- Reducing the number of footprint bits and increasing the number of data bits
- Positioning the footprint bits at different locations
- Using other modulation schemes other than amplitude modulation to encode data
- Testing the decoder with cropped regions from photos taken by real cameras

VIII. REFERENCES

[1] G. Dudek, J. Sattar, A. Xu, "A Visual Language for Robot Control and Programming: A Human-Interface Study", pending acceptance from IEEE/ICRA International Conference on Robotics and Automation, Roma, Italy, September 2006.

[2] FFT MegaCore Function User Guide, Altera Corporation, March 2001. http://www.altera.com/literature/ug/ug_fft.pdf Accessed November 30th, 2006.

[3] M. Fiala, "ARTag Revision 1: A Fiducial Marker System using Digital Techniques", NRC/ERB-1117, November 2004, 46 pages.

[2] Oppenheim, Alan V., and Willsky, Alan S. Signals and System – Second Edition. Prentice Hall. 1983. ISBN 0-13-814757-4.

FPGA Implementation of an Image Scaling Algorithm

Seth Davenport
Undergraduate,
Dept. of Electrical and Computer Engineering,
McGill University

John Dawson
Undergraduate,
Dept. of Electrical and Computer Engineering,
McGill University

Abstract-- **We present an FPGA implementation of the bilinear interpolation image scaling algorithm, suitable for use as a 'digital zoom' feature in a still camera.**

FPGA solutions are replacing ASIC and embedded software solutions currently in use due to their flexibility and falling cost, which makes our implementation attractive [5]. The bilinear interpolation algorithm was chosen because preliminary Matlab simulations suggest that it provides a good balance between image quality and computational complexity.

The implementation targets image quality at low cost. Therefore, speed is sacrificed for the sake of small chip size, in order to place the design on a cheap FPGA. However, we require sub-second response time for a 3.3 mega pixel image to provide acceptable user experience. By varying the amount of parallelism in the circuit, we can target the cheapest FPGA possible while still respecting this constraint. Image quality was tested by loading images into a VHDL test bench and saving the scaled output to a file for review.

Index Terms-- **Image Scaling, Digital Zoom, Bilinear Interpolation, FPGA.**

I. INTRODUCTION

A. Background

Many digital still cameras use so-called 'digital zoom' technology to enhance the range of their lenses. This consists of an image scaling algorithm which enlarges the image beyond the capabilities of the lens. In this way manufacturers provide greater zoom factors with cheaper optical hardware, at a cost of image quality.

Current industry implementations use either ASIC technology or embedded microprocessors to provide limited on-device image processing. However, these approaches can be too inflexible in the former case and too expensive in the latter. Our implementation is intended to address the need for a solution that is both cheap and extensible.

B. Algorithm Selection

Over the years, several algorithms have been proposed for the enlargement of digital images, with a considerable range of image quality and computational complexity. Most of these algorithms were designed for software; many do not lend themselves to hardware implementation due to their computational complexity. Preliminary research led us to choose the bilinear interpolation algorithm, since it provides reasonable quality with a minimum of computation.

C. Features

The design presented in this paper is intended to be extremely small, but also very flexible. It fits on the smallest of the Spartan IIE series FPGA's, and provides a continuous zoom range from approximately 1x to 256x. At its output, it can represent a maximum image size of 4095 by 4095 pixels or 16.7 mega pixels. It supports a 24-bit RGB colour model, and operates on a user-defined region of its input image.

D. Image Representation

Our Design works on 24-bit colour images, which are separated in to red, green, and blue channels eight bits deep (fig. 1). The same algorithm is applied on all three channels, which are recombined at the end to produce the final image.

Figure 1. *A full-colour image decomposed into red, green, and blue channels.*

E. Bilinear Interpolation

All image scaling algorithms map discrete coordinates in the output image *(i, j)* to continuous coordinates in the output image *(x, y)*. The bilinear interpolation approach does this by computing a weighted average of the intensity values of the four input pixels surrounding the point *(x, y)*, based on their distance (see fig. 2). This leads to formula 1, which gives the channel intensity for any output pixel *(i, j)* in terms of the nearest input intensities and the scale factor, *s*, where *(i, j)* are the integer coordinates of the output pixel, *s* is the ratio of input image to output image size, and p and q are the distances show in fig. 2.

$$\begin{aligned}O(i,j) &= (1-p)(1-q)I(\lfloor i\cdot s\rfloor \lfloor j\cdot s\rfloor)\\&+(p)(1-q)I(\lceil i\cdot s\rceil,\lfloor j\cdot s\rfloor)\\&+(1-p)(q)I(\lfloor i\cdot s\rfloor,\lceil j\cdot s\rceil)\\&+(p)(q)I(\lceil i\cdot s\rceil,\lceil j\cdot s\rceil)\end{aligned}$$

Equation 1

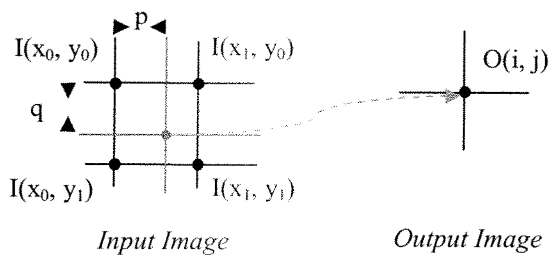


Figure 2. *Mapping of integer output coordinates to fractional locations in the input image.*

II. DESIGN METHODOLOGY

A. Design Goals

We had a variety of design goals in mind when building this project.

i. Cost

First and foremost, we needed this design to be cheap. While design cost remains rather fixed, the per-unit cost is the biggest factor in mass production. Naturally, we then decided that we should be able to implement our design on the smallest and cheapest chip possible. The line of chips we decided to use is the Xilinx Spartan 2E series, with the smallest chip being the 2S50EFT256. Xilinx claims that "the new Spartan-IIE devices give you more I/Os at much lower prices than any competing FPGA." [6]

ii. Speed

Also of great importance in our design is the speed at which it runs. Given that our target market is digital cameras, we anticipate that users will be willing to wait up to one second before becoming impatient about not seeing the zoomed image. Thus, we require that our device be able to produce an image in less than one second. Given that the standard maximum size image on a digital camera is 3.3 mega pixels, we make this the formal image size for the requirement.

B. Test Strategy

The test strategy comprised two broad classes of tests: image cases, in which several differently structured images were used as input, and boundary cases, which tested the system's response to invalid control signals. Each test case is described below; results are discussed in section IV.

i. Boundary Test Cases

TL01: *Invalid Scale Factor*
The scale factor is represented by its inverse, or the ratio of input region size to final image size. This means that a ratio of zero is not allowed, since that would imply that the output image is infinitely larger than the input region. This case tests the system's ability to detect and flag such inputs.

TL02, TL03: *Inverted Zoom Regions*
The zoom region is specified in terms of the coordinates of its top-left and bottom-right input image coordinates. However, a user could define the region 'upside down', by swapping these two values. This case tests the system's ability to detect and correct such inputs.

TL05: *Invalid Zoom Region*
Bilinear interpolation computes output pixels from blocks of four input pixels. As such, its results are not defined for zoom regions with either height or width less than two. The system should be able to detect and flag such cases.

ii. Image Test Cases

TL06: *Greyscale Image*
The purpose of this test was to ensure that the system could handle monochrome grey images. In the colour model discussed in I.(D), this situation corresponds to an image having three identical channels.

TL07-TL09: *Monochrome Red, Green, and Blue Images*
These cases test the system's ability to handle single-channel images. Such an image only has intensity values for one of its channels, the other two are set to zero. In practice, this test also ensures that our design does not erroneously mix channel data during the interpolation process.

TL10, TL11: *Minimum-Dimension Zoom Regions*
These cases test the response to zoom regions that are the minimum acceptable height and width, respectively, as discussed above.

TL12, TL13: *Coordinate Overflow*
The current implementation represents image coordinates by 12-bit vectors. This means that an image can be at most 4095 by 4095 pixels. These two test cases evaluate the result of a zoom region and scale factor that would create an image larger than this.

TL14, TL15: *Baseline cases*
These cases test that the system works under normal operating conditions, and provide a basis for image quality comparison.

C. Test Bench Design

One of the more challenging aspects of the project was finding a way to simulate the design with full image data. We envision our module as having access to an external memory unit from which it can fetch input pixels and to which it can save its results. Thus for testing purposes, we decided to implement a test bench capable of simulating such a random access memory unit. As may be seen in fig. 3, the test bench consists of a non-synthesizable memory simulator unit and some Matlab routines for pre- and post-processing of test bench data.

The memory unit is simulated using the file I/O capabilities of VHDL. Unfortunately, the `std_logic_textio` package is extremely limited in scope, and provides only for sequential access of ASCII text files. Thus, it was necessary to write Matlab routines to decompose popular image formats (JPEG, PNG, etc.) into their RGB channel information and translate it into files that could be read by VHDL. Matlab was chosen because it has very powerful image processing features built-in; this allowed us to focus more on the VHDL models without being overwhelmed by the details of image encoding formats.

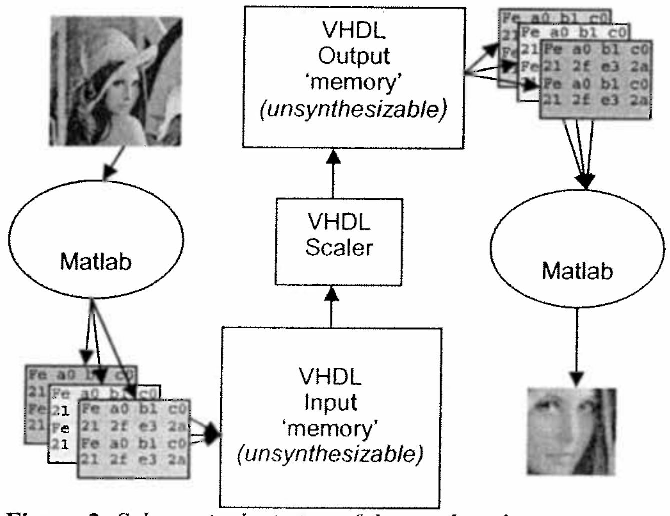


Figure 3. *Schematic depiction of the test bench.*

III. DESIGN DESCRIPTION

A. Data Path

i. Overview

As indicated in section "I. D. Image Representation", we compute the new intensity of a pixel based on the intensities of the pixels around it and the relative distance the new pixel to each of the old pixels. We could then represent the algorithm as demonstrated in the following Matlab code:

```
function [OUT] = myzoom(IMG, x0, y0, x1, y1, Sinv);

Rx = x0;
Ry = y0;
iout = 1;
jout = 1;

while Rx < x1
  Ry = y0;
  jout = 1;
  while Ry < y1

    i0 = floor(Rx);
    j0 = floor(Ry);
    i1 = i0 + 1;
    j1 = j0 + 1;
    p = Rx - i0;
    q = Ry - j0;

    OUT(iout,jout,1) = (1-p)*(1-q)*double(IMG(i0,j0,1))
                     + p*(1-q)*double(IMG(i1,j0,1))
                     + q*(1-p)*double(IMG(i0,j1,1))
                     + p*q*double(IMG(i1,j1,1));
    OUT(iout,jout,2) = (1-p)*(1-q)*double(IMG(i0,j0,2))
                     + p*(1-q)*double(IMG(i1,j0,2))
                     + q*(1-p)*double(IMG(i0,j1,2))
                     + p*q*double(IMG(i1,j1,2));
    OUT(iout,jout,3) = (1-p)*(1-q)*double(IMG(i0,j0,3))
                     + p*(1-q)*double(IMG(i1,j0,3))
                     + q*(1-p)*double(IMG(i0,j1,3))
                     + p*q*double(IMG(i1,j1,3));

    Ry = Ry + Sinv;
    jout = jout + 1;
  end;
  Rx = Rx + Sinv;
  iout = iout + 1;
end;
```

Figure 4. *Matlab code implementing our algorithm.*

In order to achieve our goals of both cost and speed, we need to be very careful in implementing our design. Each of these goals applies constraints on our design in the following ways.

- Cost

The need to conserve space means that we have to both minimize the amount of hardware we use and maximize the usage of all the hardware involved. To achieve this, we limit our design to a single ALU unit, in which intermediate values are fed back into a register file and re-entered into the ALU for future computations. Also, given the nature of the computations being done, during each pixel computation, there is idle time in which we are waiting for the results of certain multiplications. To maximize the usage of our hardware, we actually compute two pixels at a time, interleaving their computations, as shown with the multiply unit in Figure 5. While it would also appear that computing two pixel values at once would require twice the number of registers used to store intermediate values, it can be noted what with our design, this is not necessary. As each intermediate value is finished being used, it is overwritten by the value for the second pixel. In this way, we are also maximizing the use of the register file, and thus saving space.

Cycles	Multiply		Output
1-4	Pixel 1		
5-9			
10-13	Pixel 1		
14-17	Pixel 1		
1-4		Pixel 2	
5-9	Pixel 1		
10-13		Pixel 2	
14-17		Pixel 2	=> Pixel 1
1-4	Pixel 3		
5-9		Pixel 2	
10-13	Pixel 3		
14-17	Pixel 3		=> Pixel 2
1-4		Pixel 4	
5-9	Pixel 3		
10-13		Pixel 4	
14-17	•	Pixel 4	=> Pixel 3
1-4	•		
5-9	•	Pixel 4	
10-13			
14-17			=> Pixel 4

Figure 5. *Overlapping pixel computations*

- Speed

The need for speed means that our overall algorithm must be kept as quick as possible. The design indicated above requires 16 multiplications, 16 additions, and 4 requests to memory. To cut down on the time required to complete all these values, the ALU is built so as to be able to perform both an addition and a multiplication at the same time. Also, the multiplier is pipelined, to maximize its throughput. Altogether, including multiplication delays, the whole process takes 34 clock cycles to compute. Thus, by interleaving two pixels, we can produce a pixel every 17 clock cycles. Also of high importance in determining speed is the width of the data. The wider the data, the longer each multiplication will take and the slower the clock rate will have to be to allow the adder to complete its addition. We decided upon an eight bit multiplier and a twenty bit adder. The reasons for this are that the channel intensities are all eight bits, and by making our fractional values (p, q, etc.) eight bits, we minimize rounding error, so in a picture with intensities ranging from 0 to 255, there is not noticeable difference. Also, a twenty bit adder is needed to compute output pixel locations. The twenty bits is divided into 12 bits for integer digits, and eight bits for the fractional part. This is required, because if you remember from Figure 2, we need to represent real locations that have both an integer part that can be as large as the image and a fractional part that is eight bits. With twelve integer bits, we can have a maximum width and height of 4096 pixels. Given that digital cameras usually follow the standard ratios for picture sizes, this allows for pictures up to 4096 x 3072, which corresponds to a 12.5 mega pixel image and is considerable larger than we assumed the camera was capable of producing. However, using an eleven bit adder would limit the output image size to 2048 x 1536, which corresponds to a 3.14 mega pixel image, which is smaller than we want.

It should be noted that the method indicated in Figure 4 is not the only way to implement this algorithm. It is possible to compute each subsequent value of i0 and j0 by doing:

```
i0 = floor(i/S + x0);
j0 = floor(j/S + y0);
```

Figure 6. *Alternate method of determining i0 and j0.*

This method requires the use of a divider and requires the multiplier to have a width of 12 instead of 8, adding many clock cycles to or method. Thus, we decided not to use this method.

Altogether, our design ends up being quite simple, and our overall architecture is portrayed in Figure 7.

ii. Input Checker

The input checker is placed between the system inputs and the inputs of the main data path. Its function is to check the input region coordinates and scale factor for errors, correct them if necessary, and raise the appropriate output flags.

Corrigible errors concern the definition of the zoom region. This region is bounded by its top-left and bottom-right corners, denoted (x_0, y_0) and (x_1, y_1) respectively. The design of the data path imposes the constraints that $x_1 > x_0$ and $y_1 > y_0$; if this is not the case the input checker swaps the misordered values and raises a warning flag. The controller is then started and the image is scaled as normal.

Incorrigible errors involve inputs that have invalid values, or regions that are too small to be scaled (either dimension is less that two pixels). In theses cases the input checker raises an error flag and does not start the controller; the image is not zoomed and the hardware waits for the next scaling request.

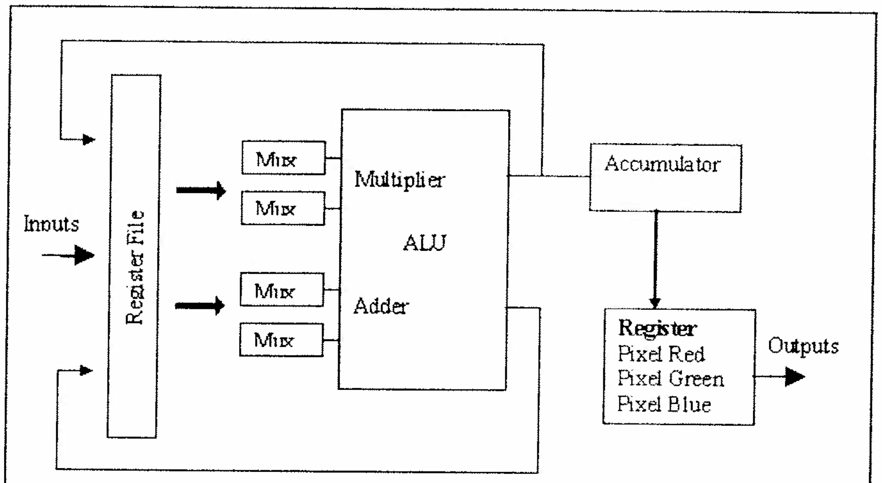


Figure 7. *Overall design*

iii. Arithmetic Logic Unit

Pixel intensity values are all members of the set of natural numbers. This simplified the ALU design considerably, since negative numbers did not have to be handled. However, the scale factor and the interpolation parameters all have fractional parts. Fortunately, this image processing application does not require excessive precision; in all cases we tested, the 8-bit fixed point representation chosen was more than enough. In fact, the test output images and the corresponding high-precision Matlab-generated comparison pictures were indistinguishable to the naked eye, despite the certain presence of rounding and truncation differences in pixel intensities.

The ALU therefore supports 8-bit fixed point multiplication, and 20-bit integer addition and subtraction. This is due to the Differential Digital Analyser [3] style approach used in the scaling: multiplications of coordinate values with the scale factor are replaced by the repeated addition of a constant fraction to the coordinate values. Since our coordinates are represented with 12-bit integer and 8-bit fractional parts, the 20-bit adder is necessary.

In initial implementation of the adder as a ripple adder was extremely slow; synthesis results suggested that the circuit would have run at one third the clock speed required to meet our time constraint. This problem led us to re-implement the adder using carry-look-ahead logic. While this led to a significant speed-up, the adder is still the critical path in our design.

Finally, after recognizing that multiplication would be the most heavily-used operation in the design, we chose an eight-stage pipelined multiplier. Fortunately, we had already implemented one in a previous project [2] which was re-used without modification.

B. Control Unit

The control unit for this device is a 20 state machine. One state is the reset state, two states are used to start up the image zooming, and the next seventeen states represent each of the seventeen clock cycles used to produce a pixel. For brevity, we have not included the state transition diagram for the states though it is easy to see how each state runs sequentially. It is important to remember here that it actually takes 34 cycles to generate a pixel, but since there are two pixels being computed at once, we can overlap the computations into 17 states. While pixel 1 is running steps 1-17, pixel 2 is running steps 18-34. Then, when pixel 1 is finished its 17th state, it becomes pixel 2. For a complete diagram of the states and the operations computed in each state, please refer to the appendix.

IV. RESULTS

A. Image Output

Below are the results of selected image output tests described above. The leftmost image in each set is the input image to the scaler; the zoomed region is marked in red. The middle image is the output, and the rightmost image is a reference image generated by Matlab.

As can be seen, the Matlab images are identical to those produced by our system. Image compositing reveals very slight differences, due to rounding errors and the lesser precision of our system, but these are so small as to be invisible to the naked eye.

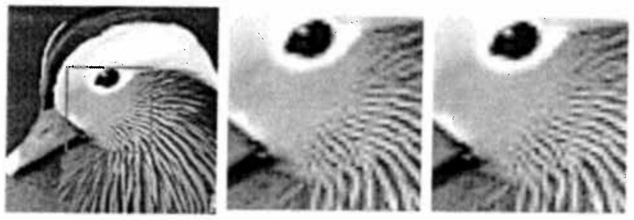


Figure 8. *Test case TL06 images. Zoom region: (30, 30) (75, 75). Zoom factor: 2x.*

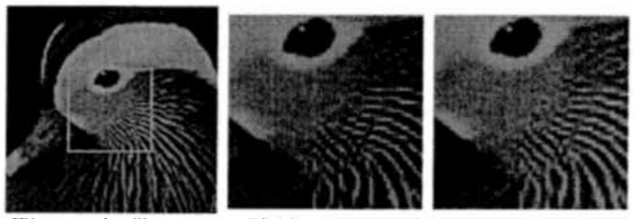


Figure 9. *Test case TL07 images. Zoom region: (30, 30) (75, 75). Zoom factor: 2x.*

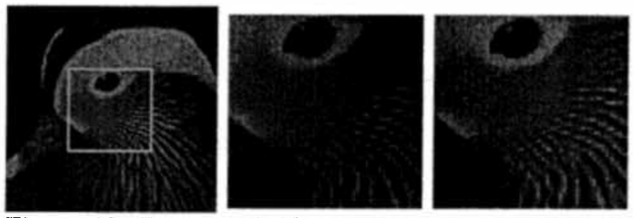


Figure 10. *Test case TL08 images. Zoom region: (30, 30) (75, 75). Zoom factor: 2x.*

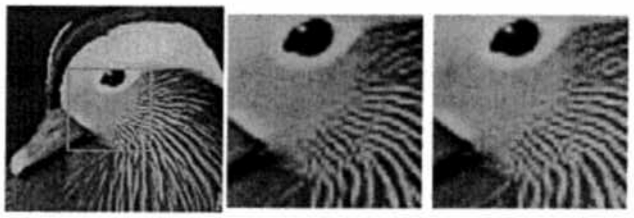


Figure 11. *Test case TL09 images. Zoom region: (30, 30) (75, 75). Zoom factor: 2x.*

Figure 12. *Test case TL10 images. Zoom region: (0, 0) to (7, 1), zoom factor: 16. Note that the input image has been shown enlarged for the sake of clarity.*

Figure 13. *Test case TL11 images. Zoom region: (0, 0) to (1, 7). Note that the input image has been shown enlarged for the sake of clarity.*

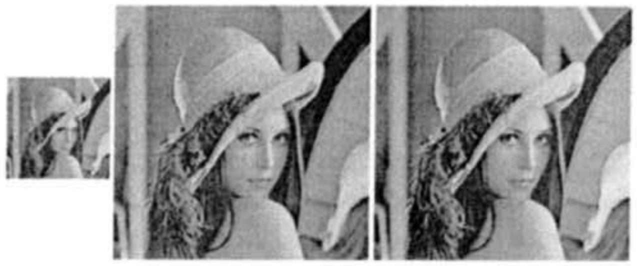


Figure 14. *Test case TL14 images. Zoom region: (0, 0) to (511, 511), scale factor 2.5.*

B. Boundary Cases

Simulation traces representing several of the boundary cases shown above may be found below. These show the various warning flags associated with invalid input values.

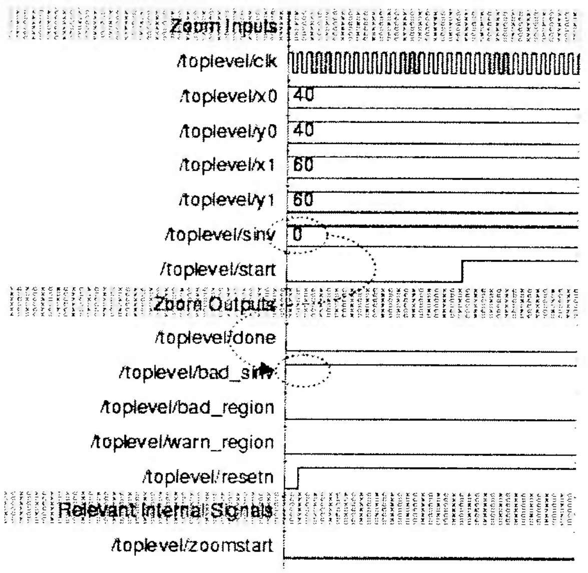


Figure 15. *Simulation trace for test case TL01. The SINV input is invalid. This causes the BAD_SINV error flag to be raised, and prevents the zoom unit from starting (ZOOMSTART stays low) even though the START input is set high.*

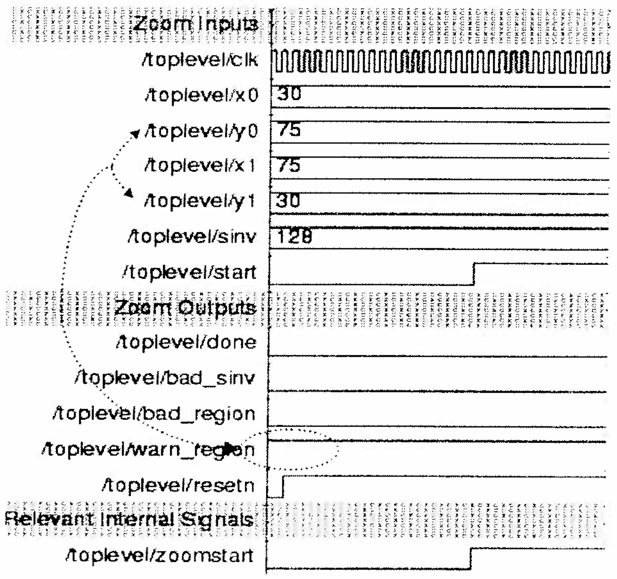


Figure 16. *Simulation trace for test case TL02. The* y *coordinates for the region specification are inverted. As a result, the WARN_REGION output flag is raised to show that the error has been corrected. ZOOMSTART is asserted when the START signal is given, which tells the scaler to operate as normal on the corrected region values.*

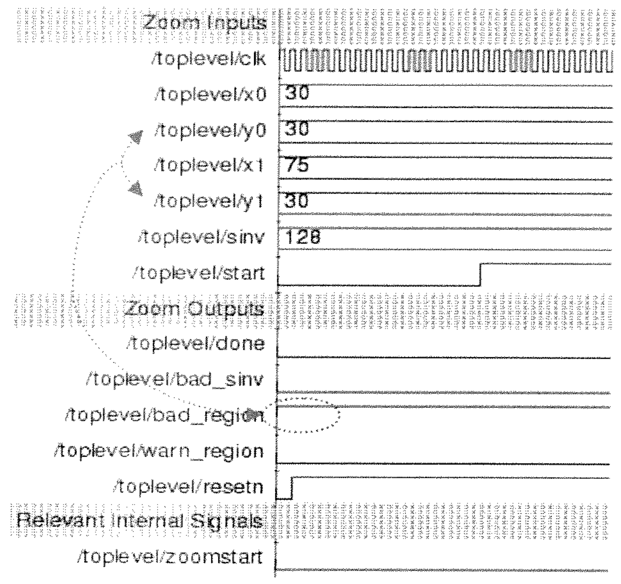


Figure 17. *Simulations trace for test case TL05. The zoom region has been defined to have a height of 0, which is an incorrigible error. As a result, the BAD_REGION error flag is raised and the scaler is not started.*

V. ANALYSES

A. Size Evaluation

As per our first goal, we required that the design be small so it can fit on a cheap chip. This goal was achieved as we fit our design onto our target chip without any real problems, using a little over half the chip (see Figure 18).

```
**********************************************
Device Utilization for 2s50eft256
**********************************************
Resource              Used   Avail   Utilization
----------------------------------------------
IOs                    158    178     88.76%
Function Generators    788    1536    51.30%
CLB Slices             394    768     51.30%
Dffs or Latches        650    2070    31.40%
----------------------------------------------

         Clock            : Frequency
     ------------------------------------
         CLK              : 39.0 MHz
```

Figure 18. *Synthesis results*

B. Performance Evaluation

Our second goal was high speed. Namely, we required the ability to compute a 3.3 Mega pixel image in less than one second. In this respect our design both fails and passes. The maximum clock rate of our design is 39MHz, which gives us an image computing rate of

$$\frac{39MHz}{17\,{}^{Hz}\!/_{Pixel}} = 2.3\,{}^{MegaPixels}\!/_{s}$$

Equation 2. *Device speed*

While we can see this is lower than our target rate, it should be noted that our design is scalable. By simply dividing up the zoomed image into pieces, it is possible to have several of our units running in parallel. Figure 19 demonstrates the method for connecting the units together.

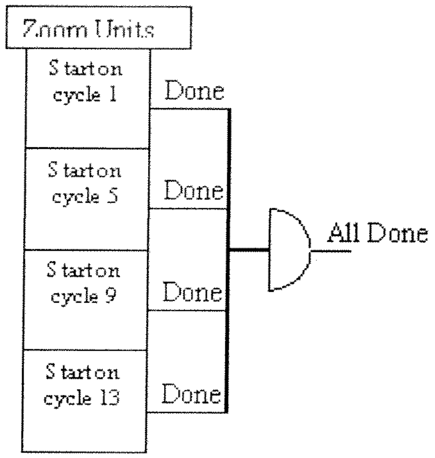


Figure 19. *Scaling design to increase speed*

While we have not implemented this interconnecting hardware, it would be incredibly simple to do, needing only a set of shift registers (to divide the image into regions – either into 2 pieces or 4 pieces), an adder to compute output pixel locations, and a simple state machine to get each unit running 4 clock cycles offset from each other. The reason this is possible is that each unit runs independently from each other, with the only time constraint being that there are memory requests. Given we have a seventeen clock cycle system, with four clock cycles required to request the intensities of the surrounding pixels, we can run up to four units in parallel, offset so no memory requests overlap, and still only require that the memory unit be able to read one memory location at a time.

We can now see how we have both fulfilled this requirement and not. While it is possible to use several units in parallel, this new design will not fit on the smallest chip anymore. However, using a larger chip may not result in a significant cost increase. A bonus feature of this design is that due to its modularity, the user can decide which trade-off they're willing to make. One the one hand, they could simply use one unit to keep cost down, but sacrifice speed, or they can use more units, increasing the cost of the chip, but allow for considerably faster computation time.

VI. CONLUSIONS

This project allowed us to experiment with designing hardware for real world situations. We had design constraints for both size and speed, and while we met these constraints on a general level, we learnt about the nature of compromising and the power of a scalable circuit. With our scalable design, we can allow the user to decide which of the two constraints is more important, and act accordingly. It should also be noted that a VHDL implementation of a bilinear zoom function is easily done on an FPGA, and there is no need for ASICs.

VII. REFERENCES

[1] Bovik, Al (Editor) "Handbook of Image and Video Processing"
©2000 Academic Press, London

[2] Dawson, John, Davenport, Seth. Scalable Pipelined Fixed-Point Multiplier.
http://www.ece.mcgill.ca/~sdaven/papers/FX_Mult_report.pdf

[3] Hearn, Donald, Baker, M. Pauline. Computer Graphics, C Version. 2nd Ed. © 1997 Prentice Hall, New Jersey pp. 87-88.

[4] Hori, B., Bier J. "Effective Fixed Point DSP Design for Low Cost Consumer Multimedia Applications"
Oct 2003, WWW:
http://www.iapplianceweb.com/story/OEG20030615S0001

[5] "Spartan-II Product Overview" Oct 2003 WWW
http://www.nalanda.nitc.ac.in/industry/appnotes/xilinx/documents/products/spartan2/overview.htm

[6] R. Olay. "Spartan-IIE Family Grows". Retrieved Nov 2003 from the World Wide Web
http://www.xilinx.com/publications/xcellonline/xcell_45/xc_sp2e45.htm

Appendix:

The following table depicts the order of operations required to compute the output intensities of the pixels we wish to generate. In our unit, we have a separate Multiplier, Adder, and Accumulator so each of these columns can have operations running concurrently. We also need to order pixel intensities, so this operation is added as a separate column too. In our state machine, we can see that for each state, we simply need to set the signals to allow each of the operations that happen in that cycle. The first 17 cycles in this diagram represent the first run of the system, where we only have one pixel being worked on. The following sets of 17 cycles, we see that we have two pixels being computed at once. Although at first, it may seem that we would need 34 states, if you collapse the operations for pixel 1 and pixel 2 for each unit (multiply, adder, accumulator), you see that there are only 17 unique states.

Notes on abbreviations:

A: (1-p)(1-q)	I1: Intensity for pixel (x0, y0)
B: (1-q)p	I2: Intensity for pixel (x0, y1)
C: (1-p)q	I3: Intensity for pixel (x1, y0)
D: p*q	I4: Intensity for pixel (x1, y1)
Rx: x0 + p	Sinv: Inverse of the scaling factor
Ry: y0 + q	

	Multiply		Adder		Accumulator		Order Intensities	
Cycle	Pixel 1	Pixel 2	Pixel 1	Pixel 2	Pixel 1	Pixel 2	Pixel 1	Pixel 2
1	p*q		(1-p)					
2	(1-p)q		(1-q)					
3	(1-q)p		j0+1					
4	(1-p)(1-q)		i0+1					
5							I4	
6							I3	
7							I2	
8							I1	
9			Ry + sinv					
10	D*I4							
11	C*I3							
12	B*I2							
13	A*I1							
14	D*I4							
15	C*I3							
16	B*I2							
17	A*I1							
1		p*q		(1-p)	D+0			
2		(1-p)q		(1-q)	Acc C			
3		(1-q)p		j0+1	Acc B			
4		(1-p)(1-q)		i0+1	Acc A			
5	D*I4				D+0			I4
6	C*I3				Acc C			I3
7	B*I2				Acc B			I2
8	A*I1				Acc A			I1
9				Ry + sinv				
10		D*I4						
11		C*I3						

12		B*I2							
13		A*I1			D+0				
14		D*I4			Acc C				
15		C*I3			Acc B				
16		B*I2			Acc A				
17		A*I1							=> Pixel 1
1	p*q					D+0			
2	(1-p)q					Acc C			
3	(1-q)p					Acc B			
4	(1-p)(1-q)					Acc A			
5		D*I4				D+0	I4		
6		C*I3				Acc C	I3		
7		B*I2				Acc B	I2		
8		A*I1				Acc A	I1		
9									
10	D*I4								
11	C*I3								
12	B*I2								
13	A*I1					D+0			
14	D*I4					Acc C			
15	C*I3					Acc B			
16	B*I2					Acc A			
17	A*I1								=> Pixel 2
1		p*q		(1-p)	D+0				
2		(1-p)q		(1-q)	Acc C				
3		(1-q)p		j0+1	Acc B				
4		(1-p)(1-q)		i0+1	Acc A				
5	D*I4				D+0			I4	
6	C*I3				Acc C			I3	
7	B*I2				Acc B			I2	
8	A*I1				Acc A			I1	
9				Ry + sinv					
10		D*I4							
11		C*I3							
12		B*I2							
13		A*I1			D+0				
14		D*I4			Acc C				
15		C*I3			Acc B				
16		B*I2			Acc A				
17		A*I1							=> Pixel 3

An Implementation of Tomasulo's Algorithm

Brian Carrillo and Peter Levine

Abstract--**Tomasulo's algorithm is a technique used in the design of computer hardware to allow for automatic and efficient exploitation of floating-point arithmetic units [Tomasulo, 1967]. The algorithm uses specialized control hardware to maximize the number of arithmetic instructions executed in a pipelined architecture. This hardware includes "reservation stations" to buffer instruction operands and a "common data bus" to load operands into arithmetic units. In addition, the algorithm defines a register renaming scheme to avoid write-after-write (WAW) and write-after-read (WAR) hazards.**

A pipeline to perform floating-point operations using Tomasulo's algorithm will be designed. This will include adders and multipliers as well as reservations stations and the common data bus. Each module in the pipeline will have its own control circuitry and will transfer data via the common data bus. All hardware will be designed and simulated using VHDL.

Implementation of Tomasulo's algorithm is interesting and challenging because it requires the designer to have a thorough understanding of advanced pipelined architectures and how these are constructed at the register transfer level. In addition, since Tomasulo's algorithm is currently used in numerous architectures like the PowerPC, it demonstrates the elegance and efficiency of modern computers.

The design of the pipeline based on Tomasulo's algorithm will be evaluated using a set of real assembly instructions. The type and ordering of these will be chosen to demonstrate that the pipeline operates correctly and that the number of wasted clock cycles is minimized.

Index Terms—**Tomasulo's algorithm, pipelining, floating-point, register renaming, scoreboarding.**

I. Introduction

When IBM built its System/360 Model 91 computer in 1967, the designers found that simple serial execution of floating-point instructions took too long to complete and fast execution times using universal floating-point execution units were difficult to achieve. As a result, an engineer named Robert Tomasulo devised a method of executing floating-point instructions concurrently using multiple instruct ion units. This scheme, which is known as Tomasulo's algorithm, was able to overcome the long floating-point delays and memory accesses associated with the Model 91. Today, Tomasulo's algorithm is used in numerous computer architectures such as the PowerPC and MIPS.

Tomasulo's algorithm uses special hardware buffers called "reservation stations" and a "common data bus" (CDB) to implement scoreboarding and register renaming schemes. These eliminate write-after-read (WAR) and write-after-write (WAW) hazards during instruction execution. The avoidance of these hazards reduces the number of stall cycles executed by the processor, effectively decreasing execution time.

This report explains an implementation of a pipelined computer architecture that is based on Tomasulo's algorithm. A brief description of the algorithm is presented followed by detailed descriptions of each functional unit in the pipeline. Simulations of the individual units are shown and explained in order to verify that each operates correctly. Next, the components are combined to form the entire pipeline and its operation verified. Sections discussing how design issues were resolved, design improvements, and the results of logic synthesis of the pipeline are also included.

It must be noted that the arithmetic execution units incorporated in this design perform integer operations only. Although Tomasulo's algorithm was originally intended to speed up pipelines that contain floating-point execution units, the out-of-order execution as well as the scoreboarding and register renaming schemes used in the algorithm can still be demonstrated. In addition, the implementation of floating-point execution units adds a great deal of unnecessary complexity to the design.

All functional units were designed using VHDL. Simulations were performed using Altera's MAX+plus II synthesis and simulation software.

II. The Algorithm

Tomasulo's Algorithm is composed of three main steps:

1. Issue: An instruction is fetched from the instruction queue and an empty reservation station is found. After this, data is routed to the appropriate reservation station. If an empty reservation station cannot be found, the instruction queue is stalled.

2. Execute: The operation stored in the reservation station is performed by the appropriate execution unit if the operands are available. If the latter condition is not satisfied, the reservation station monitors the CDB for the required operands.

3. Write-Back: Results from execution units are written to the CDB and subsequently latched by reservation stations and registers.

Both authors are Computer Engineering students at McGill University, Montreal, Quebec.

III. Instruction Set Architecture

The pipeline based on Tomasulo's algorithm was designed to operate using a register-register instruction set architecture (ISA). All instructions are 36 bits in length and are composed of a 4-bit opcode, 16-bit address for operand A (also known as the "sink" of the instruction), and 16-bit address for operand B (also known as the "source" of the instruction). Table 1 shows the opcode for each instruction.

Opcode	Instruction
0000	Add
0001	Subtract
0010	Multiply
0011	Divide
0100	Load
0101	Store

Table 1: Opcode Definitions

Examples of instructions using this architecture are:

Load A, B
Store A, B
Add A, B

In the Load instruction, operand A specifies the address of the register used to store the data from the memory location specified by operand B. In the Store instruction, data found in register A is stored in the memory location specified by operand B. When the Add instruction is issued, the data in registers A and B are added and the sum is stored in sink register A.

IV. Functional Units

This section describes how each functional unit in the pipeline was designed to operate. A figure showing how all blocks are interconnected to form the entire pipeline is included in Appendix A.

A. Decoder

Upon receiving an instruction from the instruction queue, the decoder examines the specified opcode. It then checks the busy lines of the reservation stations associated with the execution unit that will perform the specified instruction operation. If the decoder finds a station that is not busy, it sets its operand lines with the register addresses of operands A and B. The decoder also notifies the floating-point register unit as to which reservation station the data associated with the operands must be sent. This information is latched by the floating-point register unit and is subsequently used to retrieve the data at the appropriate addresses. In addition, the opcode itself is routed to the reservation station and floating-point register unit by the decoder.

In the event that all the reservation stations associated with a particular execution unit are busy, and the next instruction in the queue requires use of that unit, the decoder stalls the instruction queue. During the first clock cycle of a stall, the decoder stores the next instruction from the queue and only broadcasts the required operation and register addresses when one of the busy signals from the group of reservation stations is deasserted.

B. Floating-Point Register Unit

The floating-point register unit is composed of sixteen, 5-bit tag registers and the same number of 16-bit data registers. Its functions are to provide the reservation stations with valid data and latch data from the CDB when the arithmetic units have completed calculating the results of an instruction.

When the register unit receives the reservation station and operand addresses from the decoder, it stores the reservation station address in a register associated with the sink operand. This storage unit is known as a "tag register" and the address of the reservation station it holds is called a "tag". At the same time, the register unit sends the tags associated with operands A and B, as well as the data stored at these register locations, to the appropriate reservation station. The tags sent to the reservation station can be set to either of the values described below depending upon the state of the pipeline:

1. The tag will be the address of the reservation station that will produce a result to be written into the registers. In this case, the reservation station will ignore the data sent from the floating-point register unit.

2. The tag will be a special value to indicate that the data currently being sent to the reservation station from the register are valid. When this occurs, the reservation station should latch the data and indicate that it is ready to be processed by the appropriate execution unit.

C. Reservation Station

The reservation station performs the following operations in the order shown below:

1. The station monitors the decoder's address line and operator output. It latches the opcode and sets its busy bit when the address from the decoder matches its own.

2. The station then checks the address lines of the registers. When this address matches that of the reservation station, the station latches both the tags and data for the source and sink operands.

3. The reservation station compares the CDB address lines with the stored tags. If there is a match, the station stores the data from the CDB data lines into the appropriate data register. The reservation station then replaces the tag with a special bit pattern indicating that the data is valid.

4. When both operands in the reservation station contain valid data (i.e., both tags contain valid bit patterns), the station asserts its ready line.

5. The reservation station then clears its busy bit when its associated execution unit completes the operation.

D. Common Data Bus (CDB)

The CDB receives data from the arithmetic units and carries data to the floating-point register unit. It also accepts address tags from the execution units and brings them to the reservation stations.

The CDB is composed of an arbiter and a multiplexer which grants the bus to any execution unit that needs to write to the bus. The arbitration scheme is such that the load/store unit has the highest priority over the bus, multiply/divide has the second highest, and the add/subtract unit has the lowest priority.

E. Execution Units

Two arithmetic units and a single load/store unit have been implemented in the pipeline. The first arithmetic unit performs addition and subtraction while the second performs multiplication and division. Each unit has three reservation stations associated with it.

When a reservation station has received the necessary data to perform its operation, the station notifies the appropriate execution unit by asserting its ready line. Once the execution unit sees this, it latches the data from the reservation station and determines the result. Once granted permission from the arbiter, the execution unit places the result of the instruction and the address of the reservation station from which the instruction was received on the CDB.

V. Implementation of Key Features of Tomasulo's Algorithm

A. Out-of-Order Execution

In order to exploit instruction-level parallelism (ILP), multiple execution units must be installed in the pipeline. However, because of the inherent latencies in different instructions, sequential instructions may not necessarily complete in the order in which they were issued. This out-of-order execution of instructions can cause such data hazards as WAR and WAW.

Out-of-order execution was implemented by creating multiple execution units that perform different operations. Since these units are independent of one another, they can execute instructions concurrently.

B. Scoreboarding

Scoreboarding is a technique that allows instructions to be executed out of order when no structural hazards are present in a pipeline. In Tomasulo's algorithm, the decoder and floating-point register unit provide the scoreboarding feature. During an instruction issue, the decoder and floating-point registers broadcast a reservation station address and data to all reservation stations. Each station compares its own address with the broadcasted address and then decides whether or not it should store the data.

C. Register Renaming

Perhaps the most interesting feature of Tomasulo's algorithm is its register renaming capabilities. In a register renaming scheme, register names are dynamically allocated in order to avoid data hazards. In this implementation of Tomasulo's algorithm, registers are renamed when the floating-point register unit issues data to the reservation stations. If the data in the floating-point registers are not valid (i.e., the registers are waiting for the completion of other instructions), the unit will issue a tag indicating which reservation station holds the instruction needed to compute the result. The process of assigning tags to register addresses implements register renaming in the pipeline.

VI. Resolution of Design Issues

This section describes some of the decisions that were made in order to solve various design problems.

A. Decoder Instruction Issuing

When the decoder receives an instruction from the instruction queue, it issues the operands to the appropriate reservation station and sends the address of this station to the floating-point register unit. However, the floating-point register unit is unable to latch the issued instruction from the decoder and output the appropriate data on the same clock cycle. As a result, more than one clock cycle is needed to issue a single instruction. Since this delay is expensive, a new scheme was developed.

In this new scheme, the decoder issues a new instruction on every clock cycle. The floating-point register unit latches the address of the reservation station to which the decoder sent the instruction. On the next clock cycle, while the decoder is issuing a new instruction, the register unit sends the data to the reservation stations. Although this design causes one extra clock cycle delay, it allows for the issuing of instructions to be pipelined.

B. Reservation Station Busy Response

There is a timing issue involving the busy line of a reservation station that has just received an instruction from the decoder. Since it takes one clock cycle for the busy signal of a reservation station to propagate back to the decoder, the decoder could send another instruction to the same reservation station if designed improperly. Therefore, the decoder has been designed to always store the address of the reservation station to which it has just sent an instruction. This ensures that new information is never sent to a busy reservation station.

C. Stall Resolutions

When a stall occurs, the decoder cannot notify the instruction queue on the same clock cycle. As a result, instructions continue to be sent to the decoder from the queue. If this problem is not corrected, a single instruction would be lost on the first clock cycle of every stall because the decoder has no place to route this instruction. This is unacceptable.

In order to resolve this issue, the decoder was designed to store the instruction sent to it by the queue during the first clock cycle of a stall. Once the structural hazard has been resolved, the decoder issues the stored instruction and ceases stalling the instruction queue. As a result, no clock cycles are wasted and no instructions are lost during stalls.

D. Tag Encoding Schemes

Since there is a great deal of data circulating in the pipeline and it is the tags that determine which functional unit should latch the data, certain tag bit patterns are required to indicate that the data is invalid and should be ignored. At one point in the design of the pipeline, the bit pattern '11111' was used to indicate this. Unfortunately, this same pattern was used to indicate that the data stored in the reservation stations was valid and that the data in the floating-point registers was correct and not waiting on any other reservation stations.

In order to resolve this, three different bit patterns were used to differentiate between erroneous data, valid data in the reservation stations, and valid data in the floating-point registers. The bit patterns selected were '11111', '11110', and '11100'.

E. Execution Speed of Arithmetic Units

During initial simulations of the arithmetic units, it was found that results were computed in less than a single clock cycle. This excessive speed prevents the pipeline from taking full advantage of Tomasulo's algorithm. This issue was overcome by adding an artificial delay to each execution unit. These delays were set to match those measured in the IBM System 360/Model 91 computer.

VII. Simulation Results

This section shows simulations of the decoder, floating-point register unit, and reservation station. In addition, a simulation showing the operation of the entire pipeline is included in Appendix B.

A. Decoder

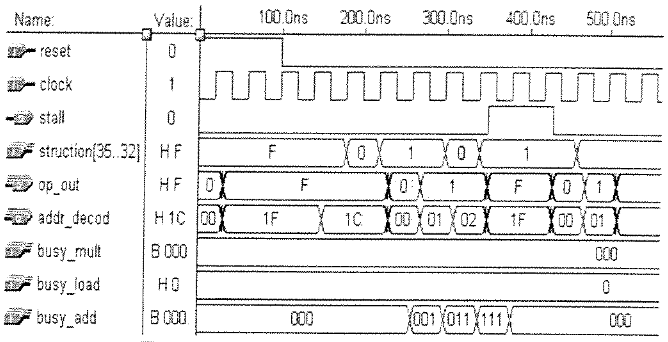


Figure 1: Simulation of decoder.

Figure 1 shows the operation of the decoder. After the 'reset' line is deasserted, the bit pattern for an add operation is specified by 'Instruction[35..32]' at 180 ns. On the next clock cycle, the decoder sets its 'op_out' line with the bit pattern for the add operation and specifies that the operator should be sent to the first reservation station ('00'). On the next clock cycle, the decoder issues a subtract instruction ('01'). Since the first reservation station is busy due to the previous add operation, the decoder sends the current operator to reservation station '01'.

The third operation is another subtract. This is sent to reservation station '02'. The next operation is an add. This causes the decoder to stall the instruction queue since there are no available reservation stations associated with the add/subtract arithmetic unit. At 380 ns, one of the reservation stations becomes free. The decoder becomes aware of this on the next clock cycle. After this event, the decoder drops the stall line and issues the saved add instruction. It then continues to issue instructions received from the instruction queue.

B. Floating-Point Register Unit

On the first clock edge in Figure 2, after the reset signal is deasserted, the floating-point registers receive an add instruction from the decoder. This is indicated as '00' on the 'OPER_IN' lines to the registers. The operands to which this operation is to be applied are '03' and '08', as shown on the 'INST_OPRND_A' and 'INST_OPRND_B' lines. Since the add operation involves a write-back to the registers, 'TAG3', the address of the operand A register, is set to '00' indicating that the result of the Add operation will be produced by reservation station 0.

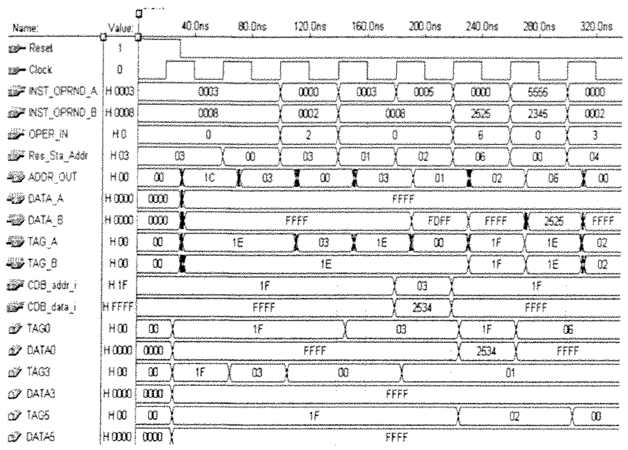


Figure 2: Floating-point register unit simulation

The next instruction occurs at 100 ns. This is a multiply operation involving operands at addresses '00' and '02'. Since the multiply operation is being sent to reservation station 3, 'TAG0' is set to '03'.

At time 180 ns, when the result produced by reservation station 3 comes back on the CDB, the floating-point registers latch this data ('2534').

Further inspection of the waveform indicates that an instruction can be processed by the floating-point registers on every clock cycle.

C. Reservation Station

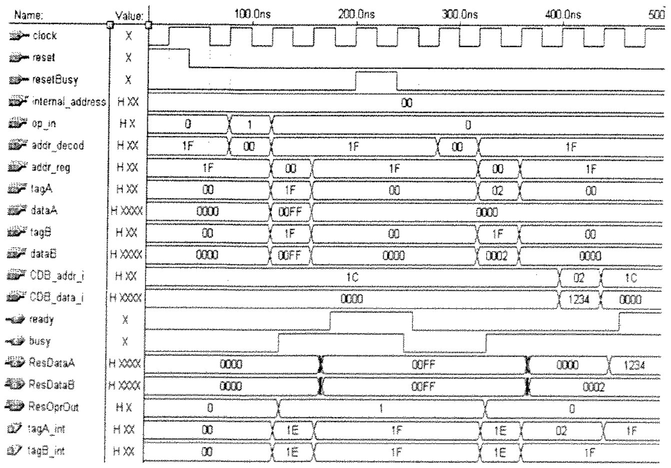


Figure 3: Simulation of reservation station.

Figure 3 shows a simulation of reservation station '00'. At time 80 ns, the decoder broadcasts a subtract instruction ('01') on the 'op_in' lines of all reservation stations and sets the address line 'addr_decod' to '00'. The reservation station shown compares its internal address with the 'addr_decod' line and because they match, the operation is stored in the reservation station. The station then asserts its busy signal on the next clock cycle.

The floating-point register unit broadcasts the address of the reservation station shown on the 'addr_reg' line as well as the tags and data associated with both instruction operands on the 'tagA', 'tagB', 'dataA', and 'dataB' lines. Since the tag lines are set to '1F', the reservation station knows that the data sent directly from the floating-point registers (i.e., 'FF00') is valid. The reservation station asserts its ready line on the next clock cycle, indicating that its operands are ready to be processed by its associated arithmetic unit. At time 200 ns, the 'resetBusy' line is asserted which indicates that the arithmetic unit has performed the specified operation and has written back to the CDB.

At time 320 ns, the reservation station receives another add instruction. However, this time, the data associated with operand A is not available in the floating-point registers. Rather, the result will come from the operation stored in reservation station 2. This is indicated by '02' on the 'tagA' line. The reservation station must therefore latch the data from the CDB after the data is computed. Note that data for operand B can be taken directly from the floating-point registers because the tagB line has been set to '1F'.

At 400 ns, the CDB address lines ('CDB_addr_I') are set to '02' indicating that the operation stored in reservation station 2 has completed and its result has been placed on the CDB. When reservation station 0 sees this, it stores the data from the CDB and asserts its 'ready' line.

D. Entire Pipeline

Appendix B shows the operation of the entire pipeline with all components connected. The instructions executed are as follows:

1. **MULT F0, F1**
2. **ADD F2, F2**
3. **SUB F3, F4**
4. **ADD F5, F5**
5. **SUB F4, F4**
6. **ADD F3, F0**
7. **ADD F3, F3**
8. **ADD F3, F3**
9. **ADD F3, F3**

The first data hazard in this instruction mix is between instructions 1 and 6. This is a WAR hazard because the ADD instruction could read the data in register F0 before the MULT instruction has had time to write to it. The subsequent data hazards are a series of WAW hazards. For example, due to the out-of-order execution inherent in the pipeline, there is a possibility that instruction 7 will write to register F3 before instruction 6 has the opportunity to do so.

Instructions 7 to 9 are a worst-case scenario for an out-of-order execution pipeline. This is because WAR and WAW hazards appear repeatedly and with the same register. In order to overcome these hazards, the register renaming scheme incorporated in Tomasulo's algorithm must be exploited. As the simulation shows, these instructions are completed successfully.

VIII. Improvements

A. *Number of Reservation Stations Per Execution Unit*

In order to reduce the number of stalls issued by the decoder, more reservation stations could be added to each arithmetic unit. This would allow more instructions of the same type to be issued without causing structural hazards. The main problem with this improvement, however, is that the complexity of the pipeline is increased and the addition of more reservation stations might not necessarily improve performance.

B. *Number of Execution Units*

Increasing the number of arithmetic units would allow for more instruction-level parallelism because operations of the same type could be executed concurrently. Although this addition almost guarantees better performance from the pipeline, the amount of hardware is greatly increased.

C. *Arbitration Scheme*

The current arbitration scheme used for the CDB is that of a simple algorithm which gives more priority to slower execution units and less priority to faster ones. A more advanced arbitration scheme could be implemented such as "first-come-first-serve". This might have the effect of increasing throughput on the CDB.

D. *Number of Common Data Buses*

Currently, all functional units are attached to a single data bus. This causes a bottleneck in the pipeline because all execution units must wait their turn before being able to write back to registers. If the number of data buses were increased such that each execution unit has its own bus, then no contention for the bus would occur and a write-back could happen immediately. The major drawback to increasing the number of data buses is that the hardware complexity of the reservation stations and floating-point registers, which monitor the buses, is increased.

E. *Decoder Issuing Scheme*

When multiple execution units of the same type are available in the pipeline, an advanced decoder issuing scheme can distribute instructions uniformly between the units. Implementing a more advanced scheme will improve the amount of instruction-level parallelism because instructions will not be waiting idle in reservation stations.

IX. Logic Synthesis and Timing

The pipeline based on Tomasulo's algorithm was implemented in VHDL. In order to facilitate reduced compilation times, a pipeline consisting of only two execution units and three reservation stations per execution unit was synthesized and simulated at a time. Therefore, the pipeline either contained one of the arithmetic units and the load/store unit or two arithmetic units.

Prior to synthesis of the VHDL code, it was estimated that approximately 750 flip-flops would be required by the design. This value was obtained by adding the number of bits that each functional unit in the pipeline must store and the number of outputs from each unit. This value also assumed that only two execution units and a total of six reservation stations were present in the pipeline. Synthesis using Altera's MAX+plus II software indicated that the design required a total of 829 flip-flops and 3986 logic cells. In addition, the design was successfully routed to an Altera FLEX10K100EBC356-1 field-programmable gate array (FPGA).

Timing analysis revealed that the longest delay path in the pipeline was 9.8 ns and that this occurred from the clock to CDB multiplexer outputs. Therefore, the highest frequency at which the pipeline can be operated is approximately 100 MHz.

X. Conclusion

In conclusion, a pipeline based on Tomasulo's algorithm was successfully implemented. As the simulations show, the pipeline was able to overcome WAR and WAW data hazards in a sequence of instructions. In addition, the pipeline was able to issue one instruction on every clock cycle. This pipeline based on Tomasulo's algorithm can easily be incorporated into a larger pipeline which would perform all other necessary operations, such as branching and jumping, to make an entire CPU.

XI. References

Hennessy, J. L. and D. A. Patterson [1996]. Computer Architecture: A Quantitative Approach, Morgan Kaufmann, San Francisco.

Hwang, K. [1993]. Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw-Hill, New York.

Tomasulo, R. M. [1967]. "An efficient algorithm for exploiting multiple arithmetic units", IBM J. Research and Development 11:1 (January), 25-33.

Digital Audio Amplification Using Sigma Delta Modulation and Bit flipping

Jakub Dudek, David Lamb, and Frédérick Chalifoux

Abstract— **This project will implement the front end of a digital audio power amplifier. It will consist of a SPDIF (CD data) decoder, an interpolating moving average filter, a 5th order digital to analogue sigma delta modulator with bit-flipping. The SPDIF decoder reads 32 bit words from a stereo CD and outputs 20 bit words into the moving average oversampling filter. The filter oversamples the data 64 times and sends it to the sigma delta modulator. The modulator encodes the original signal using only 1 bit, given that its input is properly oversampled. The pulse repetition frequency (PRF) in the output bit stream is reduced using a method called bit-flipping.**

True digital amplification is a relatively new technology and there are very few models available on the market. Digital amplifiers present several advantages over analogue amplifiers like energy conversion. They attain power efficiency levels of 90% and thus require smaller power supplies and no heat sinks. The realization of a working prototype is well within current market technologies.

Each module will be tested separately in Altera MaxPlusII using vector files that represent relevant input waveform. An output stage for the amplifier will also be simulated in Matlab, in order to perform different analysis (FFT, SNR) on the signals generated by the front end.

Index Terms – **Digital power amplification, one-bit DAC, power DAC, bit-flipping, SPDIF, AES3, sigma-delta modulator, PRF**

I. Introduction

The theory behind digital audio power amplifiers has been developed several years ago. However, only recent developments in digital signal processing technology have allowed the hardware implementation of high quality amplifiers. The main idea behind digital amplification is the conversion of a pulse-code-modulated (PCM) signal directly to an analogue signal without the need of intermediate analogue amplification. It follows that the heart of a digital amplifier resides in the conversion of the PCM signal to an oversampled single-bit stream that controls a power switch. This high voltage version of the one-bit pulse stream is then low-pass filtered in order to produce an amplified version of the original analogue signal across the loudspeakers.

The one-bit conversion process presented here is known as sigma-delta conversion. Despite their high linearity, sigma delta modulators are not well suited for power switching because of their high output average pulse repetition frequency (PRF). To overcome this problem, a technique called bit-flipping was discussed in [1] and is implemented here.

In this paper, we present a brief overview of digital amplification using bit-flipping sigma-delta modulators along with design requirements. The implementation was done in VHDL and synthesized for an Altera FPGA.

II. Design Methodology

A top down design approach was used to decompose the overall system into low-level and smaller modules. The reference paper [1] only provided general guidelines on the design and implementation so thorough Matlab simulations were performed in order to go from the general specifications to the design to be implemented.

Figure 1 shows a block diagram of the front end of a bit-flipping sigma-delta modulating power amplifier. The input consists of the digital output of a CD player and the output is the one-bit modulated signal that would be used to control a power switch at the output stage of the amplifier. Following the high-level abstraction of the system, an outline of the different blocks is given below:

1) *Flexible decoder for the digital output of a CD player*
2) *64 times moving average interpolation filter*
3) *5^{th} order sigma-delta modulator*
4) *Bit-flipping controller unit*

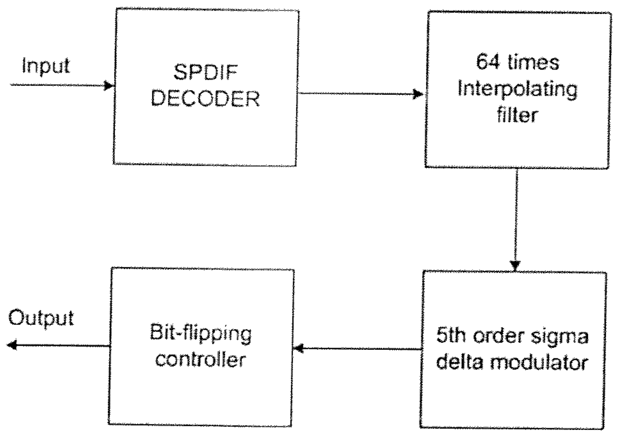

Input
SPDIF DECODER
64 times Interpolating filter
5th order sigma delta modulator
Bit-flipping controller
Output

Fig. 1. Block diagram of the front end of a digital audio amplifier

Fig. 1. Block diagram of the front end of the digital audio amplifier

Each block will be designed, implemented and tested separately in MaxPlusII. The final design entity will consist of those four components connected together.

III. Theory and Design

A. SPDIF Decoder

Music is digitally recorded on a CD at a sampling frequency of 44.1kHz. Every sample is then quantized, (mapped to a word of given length). Typical CD players perform digital to analogue conversion before outputting the signal. In the past years, many commercial units have incorporated a *digital out* output to allow external digital to analogue conversion. The *digital out* outputs the binary words encoded during original signal sampling. The format of this output is either one of two standards: SPDIF or AES3. These two formats are almost identical and differ only in the control portion of their code. AES3 contains more information about the recording details (clock precision) and is mostly used in professional recording environments. SPDIF is widely used in mass-market applications, such as in CD players and computer sound cards.

The SPDIF format is a three-layer structure: bit layer, frame layer and block layer. The bit layer defines how the bits are encoded. Each data bit is coded with two code bits so that the polarity of the SPDIF signal changes with every data bit as can be seen in figure 1. This scheme is known as bi-phase coding (or Manchester coding) and it allows for a DC-free recording. A '1' is encoded either as "01" or "10" and a '0' is encoded either as "00" or "11". Figure 2 explains the polarity change between data bits.

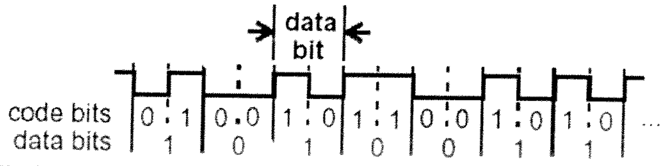


Fig. 2. Bi-phase coding: every data bit is represented using two code bits

The Frame layer indicates the architecture of the signal. There is one frame per sampling period, meaning that the frame clock runs at *fs* (sampling frequency). A frame consists of a 32-bit subframe per channel thus requiring a data clock of 64 *fs*. A subframe (shown in figure 3) is defined as follows:

1) Bits 0 to 3 are known as the preamble and indicate the beginning of a new block. They also specify the subframe's channel.

2) Bits 4 to 7 are set to zero unless 24-bit resolution is needed.

3) Bits 8 to 27 correspond to the audio sample itself. If the samples are 16bits wide as in a CD, the first four bits are set zero.

4) Bits 29 and 30 are the user and channel status respectively.

5) Bit 31 is used as a parity bit.

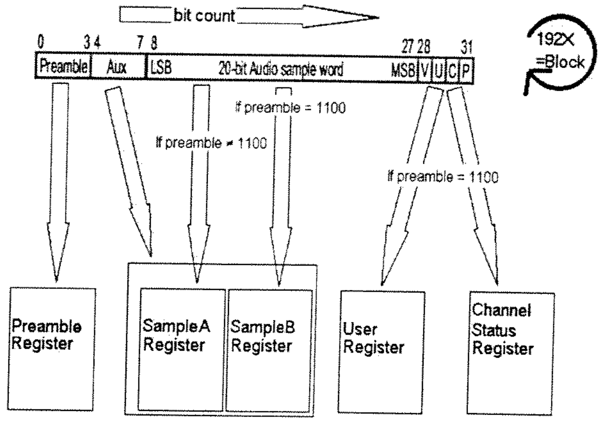


Fig. 3. SPDIF subframe

The channel status bit is extracted from 192 consecutive frames to form a 192-bit word. The same is done with the user bit. The processing of these two 192-bit words is the main difference between AES3 and SPDIF. In AES3, the user bits have no standard definition (for example they can be used for labelling in asynchronous formats) whereas in SPDIF, they contain the CD subcode. The subcode contains information about track length and limits. The channel status bits contain useful information about the system. For example channel status bits encode information regarding the nature of the signal and copyright information.

B. Interpolating Filter

Data coming from the SPDIF decoder has to be oversampled so that the sigma delta modulator produces an accurate representation of the input signal. It has been shown in [2] that an oversampling rate of 64 produces good results in audio applications.

C. Sigma Delta Modulation

The sigma delta modulator is a circuit that translates a binary number into a pulse train whose duty cycle (the fraction of time that the signal is high) is proportional to the binary input. This pulse train can then be converted into an analog signal by averaging it over time with a low-pass filter.

A first-order (single integration) sigma-delta modulation encoder is shown in figure 4. The input to the quantizer is the integral of the difference between the input and the quantized output. The difference between the input signal and the output signal approaches zero and the average value of the clocked output tracks the input. There is little dc error in the output signal; the frequency spectrum of the quantizing error rises with increasing frequency (6 dB/octave). The integrator forms a lowpass filter on the difference signal thus providing low frequency feedback around the quantizer. This feedback results in a reduction of quantization noise at low (in-band) frequencies. Using a linear model of the sigma delta modulator, the noise and signal transfer functions are found to be

$$STF(z) = \left(\frac{H(z)}{1+H(z)}\right) NTF(z) = \left(\frac{1}{1+H(z)}\right)$$

In practice, the in-band noise floor level is not satisfactory with first-order SDM. Further noise shaping must be achieved with higher-order (multiple integrations with feedback and feed forward) sigma-delta modulation coders

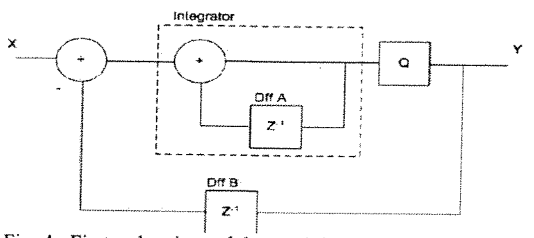


Fig. 4. First order sigma delta modulator with one integrator

The 1-bit signal can be amplified using a class D output stage, also called a MOSFET bridge or a power switch. Such an output stage never operates in the linear region except during switching time and thus draw very little current, allowing efficiency levels unattainable in traditional class A B or AB output stage.

D. *Bit Flipping*

The sigma-delta modulator presented earlier requires an oversampling rate of 64 in order to achieve high resolution in the audio band. Unfortunately, at this clock rate, the average output pulse repetition frequency (PRF) of the modulator is too high for efficient power switching in the output stage. For 44.1 KHz and 64 times oversampling, a typical range of PRF is 1.2Mhz, too high for efficient power conversion. Furthermore, since the PRF is dependant on the input amplitude, its negative effect on the power switch will be perceived as distortion.

A technique called bit flipping can be used to lower and regulate the PRF. The concept of bit flipping is quite simple: the quantizer output is inverted in a way that the 1-bit signal transition rate is lowered. A control algorithm regulates the PRF by first monitoring a change in the previous and current output. It then decides whether the current bit should be inverted according to the following two constraints:

1) PRF constraint (N): Bit flipping should occur whenever the average PRF exceeds a target PRF f_t. The average PRF can be measured by counting the number of transitions P that occurred in the last B bits and is given by

$$PRFav = \left(\frac{L * Fs}{2}\right)\left(\frac{P}{B}\right)$$

When (P/B) exceeds a target value of (1/N), a flip is required in the quantizer output. A value of N=2 yields an acceptable PRF of 370 kHz.

2) Alternation Constraint (A): To prevent the buildup of a dc error when switching, the alternation constraint regulates the number of positive (0→1) and negative (1→0) flips that occur. For example, an alternation factor (A) of 2 signifies that no more than two consecutive positive or negative flips can occur. This reduces the effect of constraint number 1 on PRF reduction.

IV. Implementation

A. *SPDIF*

The implementation of the SPDIF/AES3 decoder is comprised of two sections: a bi-phase decoder for the Bit layer and a frame and block decoder.

The bi-phase decoder was implemented using only a 2-bit counter and 2-bit shift register as can be seen in figure 5. The 2-bit counter is used as a clock divider since there is two code bits for every data bit. At every two clock cycles, the two code bits received are XOR'ed to form a data bit. A simple Finite State Machine (*FSM*) with two states ensures that the data transition occurs on the rising edge of the clock.

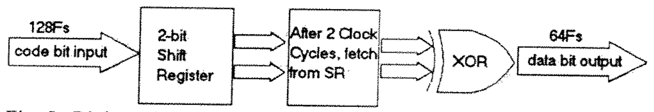


Fig. 5. Biphase decoder block diagram

The frame and block decoder uses a similar approach. In this case, a modulo-32 counter is used for the bit count and frame cycle while a modulo-192 counter is used for the frame count and block cycle. Five shift registers buffer the Preamble (4 bit), Left_channel (24 bits), Right_channel (24 bits), User (192 bits) and Channel Status (192 bits) inputs. A FSM is used in conjunction with the bit count to enable registers at different times in order to separate the incoming data properly. Each portion of the signal has it's own state, meaning that data from the Preamble state is ready in the Sample state, where Ready_Preamble is asserted.

The audio samples are declared ready in the Parity state of the Right_channel subframe for synchronization purposes. When recording, the audio signals are sampled simultaneously from the left and right channel, thus decoding should output simultaneously.

B. Interpolating Filter

The interpolating filter has two sections: the difference section and the summation section. Figure 6 shows a block diagram of the interpolation filter. The difference section computes the difference between the current and previous input and sends it to the summation section. The summation section adds the difference to the previous output to form a new output. The summation section is clocked 64 times faster than the difference section, which effectively creates 64 outputs for 1 input. The output signal's amplitude has now been scaled by a factor of 64 and thus needs to be rescaled. Every output value is right shifted 6 times to implement division by 64.

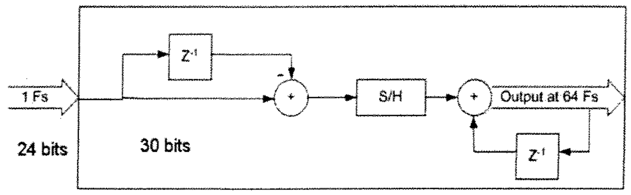


Fig. 6. Interpolating filter block diagram

All the data paths are 30-bit wide to insure that there is no overflow in computations. As the maximum difference calculated by the subtractor is $2^{24}-1$ and the modulator scales the input by a factor of 64, the difference offers a maximum output of 2^{30}-1.

C. Sigma Delta Modulation

A fifth order sigma delta modulator was implemented as shown in figure 7. More details on this topology can be found in [1] and [2]. The shifters in front of the integrators and in the local feedback paths ensure that the signals do not grow too rapidly and destabilize the modulator. Further precautions to avoid unstable behavior will be addressed.

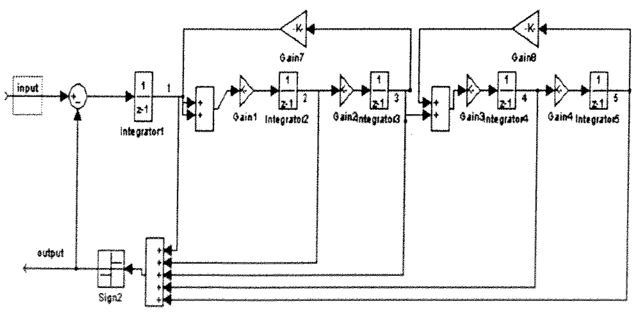


Fig. 7. 5th order sigma delta modulator

The implementation of the integrator blocks is shown in figure 8. Each integrator consists of a 24-bit adder and a 24-bit flip-flop connected in a feedback topology. 24-bit data paths are used throughout the design to avoid frequent overflow. Each integrator contains logic that resets the state of the loop filter H(z) if its internal state is larger than a value T. High order modulator can become unstable and this instability is characterized by high signal levels and a failure to code the input signal properly. Whenever the internal signal in the first integrator is greater than the threshold value T, the internal state of the whole filter is reset. Threshold values for each integrator have been found using Matlab.

The first integrator also contains hardware for the bit flipping control. Figure 12 shows the first integrator. The need for this extra adder will be discussed the bit-flipping section.

D. Bit Flipping

The overall block diagram of a Bit-flipping sigma-delta modulator is shown in figure 8. The bit-flipping is implemented in combinational logic and consists of four parts:

1) PRF monitor (figure 9): The PRF monitor was implemented as a counter (*count_N*) that counts down at every sample when no change in output is detected and counts up 2 steps when a transition occurs. When the counter output is positive, the PRF monitor requests a flip. This translates into flip_N being asserted.

2) Alternation control (figure 10): The alternation controller was implemented as a counter (*count_A*) that counts up whenever a positive flip (0→1) occurs and counts down whenever a negative flip (1→0) occurs. The current quantizer output (*Qo*) is used in conjunction with the *flip* signal to determine the sign of the flip.

3) Flip acknowledge (figure 11): Whenever the PRF monitor requests a flip, the value of the alternation counter is checked along with the sign of the current quantizer output.

4) Flipping unit (figure 12): The *flip* output of the bit-flipping controller controls a multiplexer that selects between the actual quantizer output and its inverse. For causality reasons, the bit flipping must occur outside the feedback loop of the modulator. Compensation must be made in the loop filter to take into account the change in quantizer output. Only the first integrator needs to be modified as shown in figure 13.

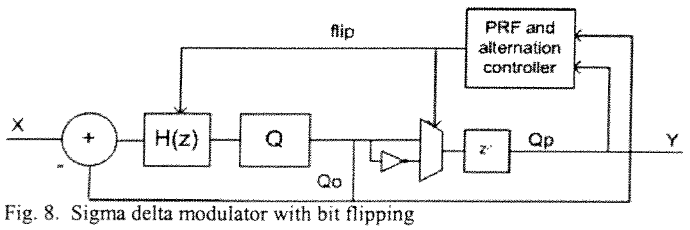


Fig. 8. Sigma delta modulator with bit flipping

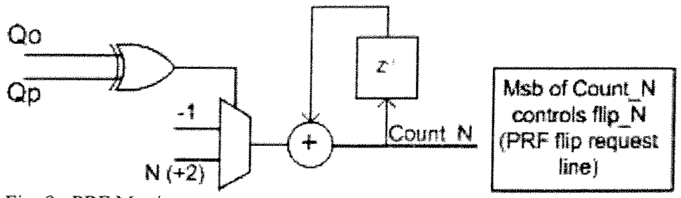


Fig. 9. PRF Monitor

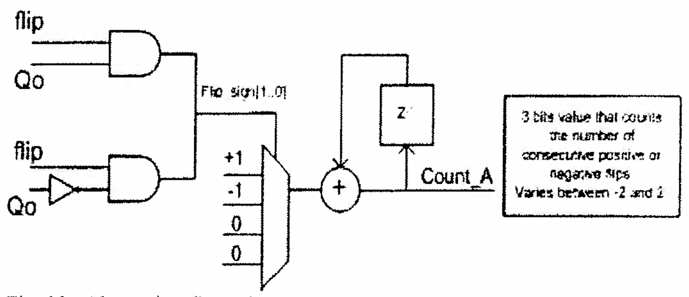

flip
Qo
Flip sign[1..0]
flip
Qo
+1
-1
0
0
+
Z⁻¹
Count_A
3 bits value that counts the number of consecutive positive or negative flips Varies between -2 and 2

Fig. 10. Alternation Control

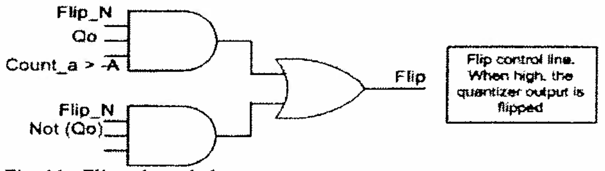

Flip_N
Qo
Count_a > -A
Flip_N
Not (Qo)
Flip
Flip control line. When high, the quantizer output is flipped

Fig. 11. Flip acknowledge

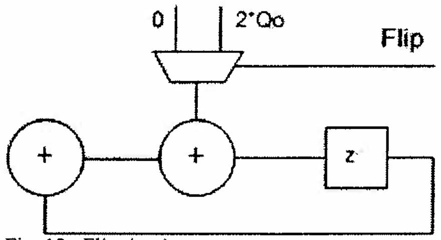


Fig. 12. Flipping integrator

V. Testing Methodology

Testing has been performed on the level of individual modules and complete system. Each module has been tested separately to initially isolate any potential problem in order to minimize debugging time during system testing. The overall system was then tested and the results are presented in the following section.

A. Module testing

1) SPDIF decoder: A vector file was used as input for the testing. A CD outputs 64 bit frames serially at a rate of 44.1 KHz. This amounts to a clock rate of nearly 6MHz for bi-phase coding. To reduce the amount of data to simulate, the bi-phase decoder was tested separately. Testing involved creating a random input and verifying the output by inspection.

A MATLAB routine was written to simulate the frame and block structure of the SPDIF/AES signals. Audio samples are generated using sinusoidal signals sampled at 44.1kHz. The results are then written to a vector file that can be imported into MaxPlusII. This input file can then be decoded and the outputs are inspected in the waveform analyzer.

2) Interpolating filter: The testing of the interpolation filter consists of taking the output of the SPDIF/AES decoder and feeding it to the filter. This provides a realistic input for the filter. In addition, other tests were performed to check the interpolation between two very close numbers, between two very distant numbers and finally between positive and negative numbers.

3) Sigma delta modulator: Thorough tests have been conducted on the sigma delta modulator with emphasis on accuracy of data coding and stability. The modulator has been sine wave tested. A sine wave was input to the MaxPlusII waveform editor by hard coding the 16-bit data values at the input. The sine wave frequency was 2757Hz sampled at 44.1kHz and oversampled 64 times. The output was exported to matlab and low pass filtered to obtain the analog sine wave. Both signals are shown in figure 13. The signal-to-noise-ratio (SNR) was calculated and a Fast Fourier Transform was performed to examine the output's frequency content. The FFT is shown in figure 17. The modulator was also run under instability conditions to test the internal reset of the integrators.

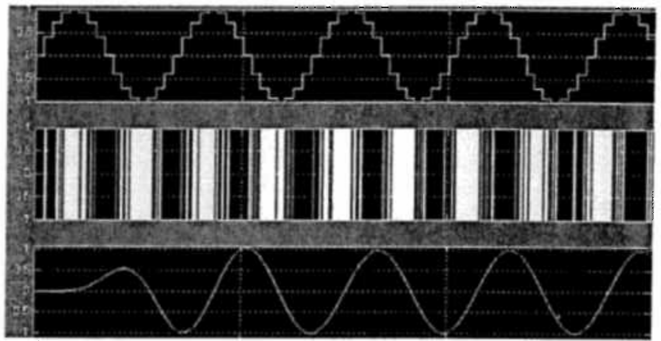


Fig. 13. The top scope shows the wave input to MaxPlusII. The middle scope shows the MaxPlusII modulated signal imported to matlab. The bottom scope shows the analog output after it has been filtered.

4) Bit flipping unit: Testing of the bit-flipping unit was done by comparing the change in PRF introduced. PRF was calculated by counting the number of output signal transitions within a given simulation time.

B. Module testing

The overall system was tested with a vector file that encapsulates a stereo SPDIF encoded signal. Each channel consisted of a sine wave. A complete waveform output is shown in figure 14 with intermediate outputs at every stage.

VI. Analysis of Results and Performance

The output waveform in figure 16 shows the output of the system simulation. From the stereo serial input bit stream, the preamble, user bits and channel bits are extracted. Both channels are also extracted in *Right_SPDIF* and *Left_SPDIF*. The oversampled output from the filter is shown in *Left_INTER* and *righ_INTER*. *Left_MOD* and *Right_MOD* are the two modulated bit flipped output. *left_flip* and *Right_flipB* are asserted whenever the bit-flipping unit flips the output to lower the PRF. The *Lesft_resetINT* and *RIGHT_resetINT* signals are asserted whenever an integrator resets itself internally to prevent instability.

The PRF and stability problems were successfully addressed in the design. Figure 15 shows the output of the modulator with and without internal resetting of integrators. As can be seen, the signal failed to be coded properly.

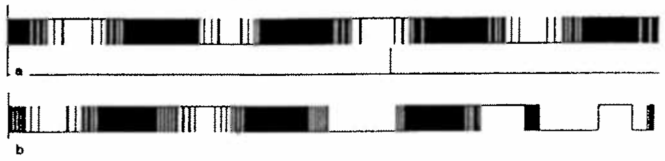


Fig. 15. The top wave is the output when the internal reset of the amplifier was enabled. The bottom wave, the internal reset was disabled, the modulator became unstable

Figure 17 and 18 show the FFT of the output signal without and with bit flipping respectively. It can be seen that bit flipping increased the noise floor by about 10 dB. It is stated in [5] that the high PRF characteristics of sigma delta modulators result in an intolerance to mismatch rise and fall times in output switching stage resulting in distortion and noise intermodulation which are far more problematic than the bit flipping noise.

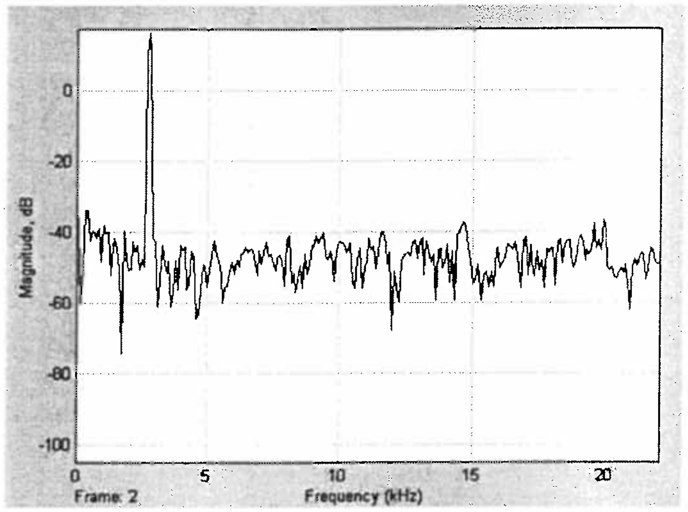


Fig. 17. FFT of the output signal without bit flipping. Noise floor is at --30dB.

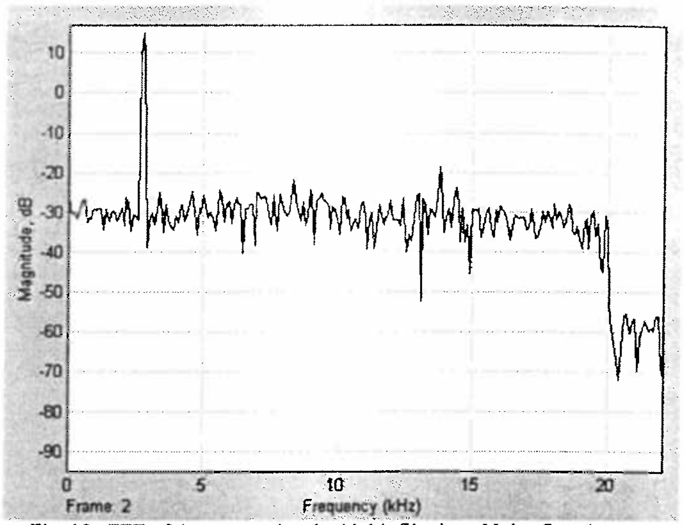


Fig. 18. FFT of the output signal with bit flipping. Noise floor is at -30dB.

The number of transitions with and without bit-flipping was 20713 and 13519 in 20ms of simulation. The average PRF was effectively reduced by 35% bringing it from 1.035MHz to 675kHz.

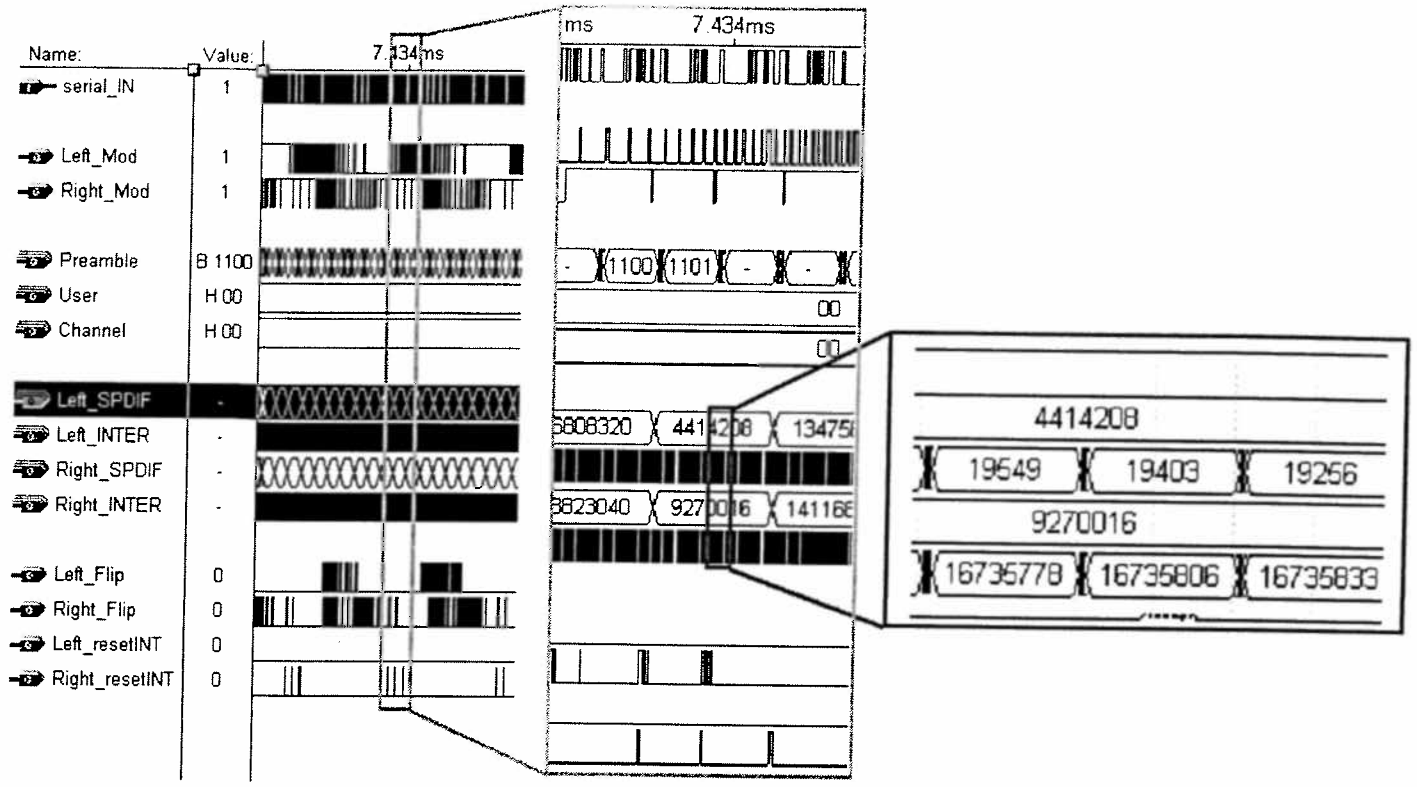


fig. 16. Output of full simulation: serial_IN is the serial input stream feeding the SPDIF decoder. Preamble, User and Channel are specific code portions of the data words. Left_SPDIF and Right_SPDIF are the 16 bit music samples. Left_INTER and Right_INTER are the stereo outputs of the interpolating filter. The zoom shows many Interpolated values for 1 audio sample (actual number is 64). Left_MOD and Right_MOD are the sigma delta modulating output for each channel. Finally, the reset and flip for each channel are acknowledged when bit flipping occurs or when internal reset occurs.

The FFT (figure 17) of our output signals show frequency peaks at the frequencies of the inputs. The signal was thus well transmitted. The noise floor however is very high, in the range of −30dB. This is far from good enough for a commercial unit. The noise floor can be reduced by implementing proper coefficient scaling in the transfer function *H(z)* of the sigma delta modulator. Details about finding optimal coefficients can be found in [2]. Time constraints has not allowed for the development of an optimal transfer function *H(z)*.

The objectives were met in a satisfactory manner. Even though this digital amplifier is not suited for commercial production, every necessary block that constitute it has been implemented and is functional. The modular design approach used makes the digital amplifier simple to upgrade and debug in the future. This design is a first step of a work in progress that will be improved in the future. It is shown in [1] that SNR of the order of 120dB are attainable. The extra noise inferred from bit flipping would be negligible at such high SNR values even though it is not with the current design.

The following table summarizes the resource usage for the implementation of a stereo front end of digital amplifier

Device Family	Flex 10k
Device	EPF10K40RC240-3

Total input pins	4
Total output pins used	59
Total logic cells used	1795
Maximum clock rate	5.81Mhz

Table 1. Hardware usage for modular exponentiation module

I. Design Challenges

Challenges were encountered mainly in the design part of the project. Documentation and relevant sources of information had to be extensively researched and authors of certain papers had to be contacted. Digital audio amplifier is a new branch of the audio field and is not very well documented. High-level designs are available but specific design had to be constructed from scratch using matlab.

Some time had to be spent figuring out the theoretical results that would have to be obtained. These were not readily found and many Matlab simulations were run to obtain theoretical results.

Understanding the S/PDIF protocol was an ambitious task as relevant sources were hard to find. Once this was accomplished, implementation was pretty straightforward. MATLAB proved very helpful in generating test benches for the decoder (12288 entries per blocks, 2822400 per second).

The testing for stability of the sigma delta modulator required creating instability conditions within a few periods of a sine wave. The output had to become unstable within only a few periods because simulating the sigma delta modulator takes up a lot of time for only a few milliseconds. The instability had to occur early so that the internal resetting could be shown to work properly.

Overall testing of the sigma delta modulator called for the design of a scheme for importing MaxPlusII output into matlab. This required creation of a table file and text file manipulation to make the file Matlab readable.

The combinational nature of the bit- flipping unit made precise timing a strong requirement. Every input had to be analyzed and decision had to be taken concerning the flipping of the current output within a clock cycle. No design for the combinational logic itself was found and thus had to be made from scratch.

II. Conclusion

An implementation of a digital audio amplifier in FPGA has been presented. The design has been shown to be a working unit but time constraints have not allowed it to attain performance suited for commercial production. The modularity of the design will allow for further development in a near future and every block will be improved until the specifications meet current technology standards. Specifically, software will be used to design an optimal modulator transfer function H(z).

Appendix

VHDL code included in the appendix

Acknowledgment

Team LCD would like to thank Jessie Gabe for helping the editing of the paper.

References

[1] A.J. Magrath, I.G. Clark and M.B. Sandler. A sigma-delta digital audio power amplifier – Design and FPGA implementation. Department of Electronic and Electrical Engineering King's College London, June 1997.

[2] A.J. Magrath. *Algorithms and Architectures for High Resolution Sigma-Delta Converters*. Phd thesis, King's College, University of London, October 1996.

[3] James C. Candy, Gabor C. Temes, *Oversampling Delta-Sigma Data Converters, Theory, Design and Simulation*. New York: IEEE PRESS, 1992.

[4] A.J. Magrath and M.B. Sandler. Digital Power Amplification using Sigma Delta Modulation and Bit Flipping. Journal of the Audio Engineering Society, June 1997.

[5] A.J. Magrath and M.B. Sandler. Digital-domain Dithering of Sigma Delta Modulators using Bit-Flipping. Journal of the Audio Engineering Society, June 1997.

[6] L. Risbo. FPGA based 32 times oversampling 8th order sigma-delta audio DAC. Presented at the 96th Convention of the Audio Engineering Society, Amsterdam, Preprint 3808, February 1994.

[7] John Watkinson, The art of digital audio, third edition, Oxford: Focal Press 2001.

[8] Leong, Choon Haw. *New architectures for High-Order Bandpass Sigma-Delta Modulation in Digital-To-Analog Converters.* Master thesis, McGill University, June 1998.

[9] Jim Thompson. (1995, June 8). Care and Feeding of the one bit digital to analog converter. [Online]. Available: http://www.ee.washington.edu/conselec/CE/kuhn/onebit/primer.htm

[10] P. Craven, "Toward the 24-bit DAC: Novel Noise-Shaping Topologies Incorporationg Correction for the Nonlinearity in a PWM Output Stage," J. Audio Eng. Soc., vol. 41, pp. 291-313 (1993 May)

[11] Ken C. Pohlmann, *Principles of Digital Audio*, McGraw-Hill, 2000

Pipelined Route Optimization Hardware For The Traveling Salesperson Problem: A Genetic Algorithm Approach

Benjamin Kuo, Angelique Mannella

Abstract—**The primary goal of this project is to determine an optimal solution of the Traveling Salesperson Problem by using a hardware implementation. The design is based on a Genetic Algorithm coded in VHDL, and later simulated on an Altera Flex10K FPGA. The nature of the Traveling Salesperson Problem makes it difficult to adhere to a true Genetic Algorithm, thus several components of the Genetic Algorithm were slightly modified. In an attempt to optimize the performance of the hardware implementation, this design is pipelined. The success of the final design is evaluated by whether the solution converges, and by how well it maps into the Altera Flex10K FPGA.**

The Traveling Salesperson Problem is representative of many routing problems found in telecommunication networks. The design will provide an alternative to software solutions, and will provide a foundation for which more complex routing hardware, such as that found in real-time communication networks, can be optimized.

Index terms—**traveling salesperson problem, genetic algorithm, VHDL, FPGA**

I. INTRODUCTION

The Traveling Salesperson Problem is one of a class of complex problems known as NP-complete problems. Non-deterministic Polynomial (NP) problems are decision or recognition problems. The decision within the Traveling Salesman Problem is, `Is there a tour of length less than L?' Formally, the Traveling Salesperson Problem (TSP) can be described as follows:

There exists a set of n nodes (cities), and a distance function

$$D : VxV \rightarrow \Re,$$

that gives the travel distance between any given pair of nodes. The goal is to find a tour of the nodes $v_1, \ldots, v_n$ such that

$$\sum_{i=1}^{n-1} D(vi, vi+1) + D(vn, v1)$$

is minimized. This is the shortest trip that visits every city exactly once. Figure I-A illustrates the idea of the TSP.

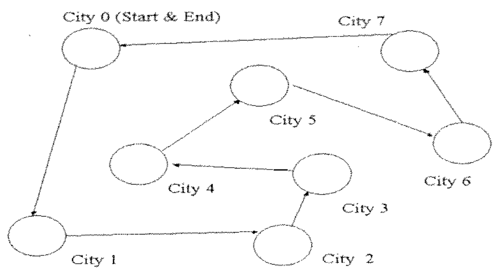


Figure I-A: TSP: Which city sequence is the shortest tour?

Solving this problem for a small number of cities is not complex. However, as cities are added to the tour, the search space grows factorially, and the computational complexity rapidly increases.

This problem has many important applications, such as route determination in transportation or telecommunication networks, thus a method of solving this problem quickly and with minimal resources is desired. One approach to solving this problem is to use a Genetic Algorithm.

Genetic Algorithms simulate the way nature uses evolution. A GA (Genetic Algorithm) contains an initial population of solutions called chromosomes. Members of this population produce new 'children' solutions, which are hopefully better solutions than the parents. This is called crossover. Mutation occurs randomly, and slightly alters the newly formed solutions. It is possible that the mutated solution will yield a better solution, that crossover could not have produced. The newly formed solutions are then subjected to a fitness test. The fitness test determines how well the solutions solve the problem at hand. In this investigation, the fitness of a solution determines whether or not the solution will be written back into the population. Only the fittest two solutions from the two parents and two offspring will survive.

Traditionally GA's have been implemented in software. However, within this paper we investigate a hardware implementation, with the hope that its advantages and usefulness will be evident.

Benjamin Kuo is an undergraduate Computer Engineering student at McGill University, Canada.
Angelique Mannella is an undergraduate Computer Engineering at McGill University, Canada.

II. OVERVIEW OF THE ENTIRE DESIGN

Several parameters are needed in the implementation a GA to solve the TSP. Firstly, solutions must be encoded in 'chromosomes'. Permutation encoding is used, as this allows every chromosome to be a sequence of cities, representing a tour. Eight cities (city 0 through city 7) are used within a tour, and each city is encoded using 4 bits (an extra bit is introduced for debugging purposes). Therefore, each chromosome is a 32 bit sequence, representing the order in which the Traveling Salesperson will visit the 8 cities. Also, the distance between all pairs of cities in the tour must be determined. The city-to-city distance is encoded in 8 bits, and stored in a ROM module, thus eliminating the need to calculate the distance.

Another necessary parameter is the population size. A population of 64 solutions is used. GA events that need to be defined include; how solutions will be selected for crossover, how crossover and mutation will occur, and how solutions will be chosen for entry into the population. Within this design, all events of the GA are pipelined. Each event within the GA is implemented as a module, and register banks exist between the modules. The population is stored in a RAM module, crossover and mutation occur in the Crossover and Mutation Module, solution fitness is tested in the Fitness Module, and chromosome removal and entry into the population occur in the Chromosome Selection and Write Back Module. Figure II-A below illustrates the entire design.

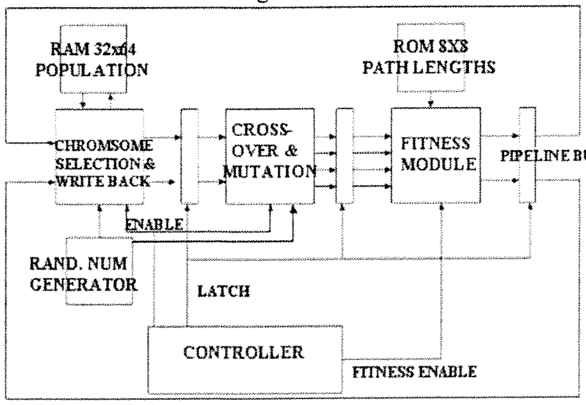


Figure II-A: Schematic of All Modules

III. MAIN CONTROLLER

The controller contains a finite state machine, which operates after a global reset signal is set. The finite state machine controls the enable, fitness enable, and latch signals. It sets the enable and fitness enable signals to '1' at specified times, allowing the modules to begin operation. When it sets the latch signal to '1', signals produced by a module are stored in a register bank. The Controller consists of 3 states: IDLE, PROCESSING and FIN state.

IDLE: The state is reset to IDLE when reset is '1'. The subreset signal is then asserted to reset all the modules. In this state, once all module control signals are set to 0 and the internal counter is reset to 0, a transition to the PROCESSING state occurs.

PROCESSING: The counter increments every clock cycle, and when it reaches 48 (the slowest stage has finished), it will proceed to the FIN state. Otherwise it will stay in the current state and increment the counter.

FIN: When this state is reached, all stages in the pipeline have finished and their outputs have stabilized. The state machine then issues the latch signal to the pipeline buffers. This causes them to update their contents with values from the stage before. The state machine will then continue to IDLE state.

IV. POPULATION OF SOLUTIONS

The initial population of 64 chromosomes was randomly generated using a program coded in C (see Appendix). The population changes with each iteration of the GA. Three components contribute to maintaining the Population of Solutions: A Random Number Generator, A RAM Module, and A Chromosome Selection & Write Back Module.

A. Random Number Generation

The random number generator plays a very important role within this design. Two randomly selected solutions are chosen from the population to generate two new solutions. Mutations also occur randomly within the newly generated solutions. Within this design, a pseudo-random number generator is used.

The random number generator is designed to output an 8 bit random number. A *seed* is required to start the generator. The seed can be selected from the random number output, after it has been initialized. The random number generator block consists of simple bit manipulations that rotate the upper 3 bits to the left and lower 5 bits to the right. To increase the randomness, a counter is used to trigger a bit flip during rotation at several different count values. This bit flip occurs in both the upper part and lower parts of the bit string. This implementation is shown in Figure IV-A.

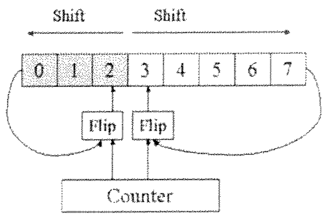


Figure IV-A: Base Random Number Generator Block

The pseudo-random number generator further increases the degree of randomness by using linear feedback. The feedback system makes use of the numbers generated, as it routes them back to the seed input. The block diagram is shown in Figure IV-B.

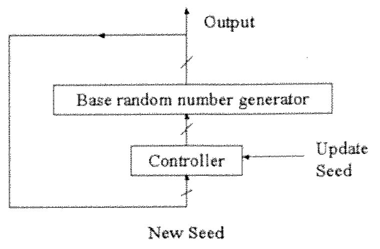


Figure IV-B: Linear Feedback Pseudo-Random Num. Gen.

At times, an enable bit will be set by the controller, that will update the seed, thereby generating a new set of random numbers.

B. *RAM Module*

The RAM Module contains the population of 64 32-bit chromosomes. This block is constructed using an Altera LPM module, which make use of the memory portion of the FPGA chip. The RAM is initialized with a memory initialization file (.mif files). For simulation purposes, a simple .mif file generator was written in C (see Appendix). This program generates random tours consisting of all eight cities. To improve access time, the RAM uses asynchronous read and write.

C. *Chromosome Selection and Write Back Module*

The Chromosome Selection and Write Back Module is enabled by the central controller. It receives random numbers from the Random Number Generator, and uses these numbers to select the addresses of two chromosomes from the population. This module also receives two chromosomes from the Fitness Module, and the addresses into which it will write these chromosomes. This Module consists of a random number filter, and a 14 state Finite State Machine.

The random number filter continually inputs random numbers into the module from the Random Number Generator, but prevents the same random number from appearing twice within three consecutive accesses. This filter is explained in more detail in section VII.

The Finite State Machine includes an abundance of states, however, many of these states are wait states. These extra states are essential, as they ensure the accuracy of reads and writes to and from the RAM module. The fourteen states are described below. Figure IV-C below illustrates the Finite State Machine.

IDLE: This is the initial state. If the enable line from the controller is set to '1' a state changes to RDATA_1.
RDATA_1: In this state the address line to the RAM module is set to a random number received from the Random Number Filter.
RDATA_2: The data stored in the address set in state RDATA_1 is retrieved, and the address line to the RAM is reset with another random number.
READ_FIN: The data stored in the address set in state RDATA_2 is retrieved. If the variable 'count' is not 3, count is incremented. Count's value determines whether a write will occur or not. If it is less than 3, the pipeline has not been initially filled.
WDATA_1: This is the first write state. Count is 3 when this state is entered. The first address to write, and the first data to write are set on the Ram address and data lines.
W_EN, W_EN2: The write enable line for the RAM module is set to '1'.
W_DIS: This state sets the write enable line to '0'.
WDATA_2: This is the second write state. It functions the same way as WDATA_1.
WRITE_FIN: This state sets the write enable line to '0' for the second time.
WAIT1, WAIT2, WAIT3, WAIT4: All these states hold the past state for one extra clock cycle.

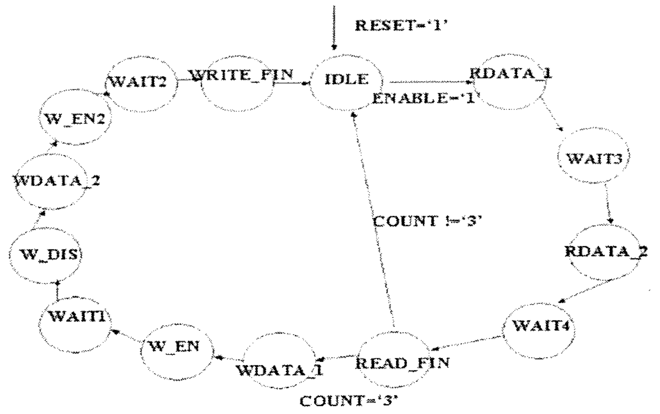


Figure IV-C: FSM of Chrm. Sel and Write Bck Module

D. *Simulation of the Population of Solutions*

The following diagrams illustrate the functioning of the Chromosome Selection and Write Back Module, the Random Number Generator, and the RAM Module.

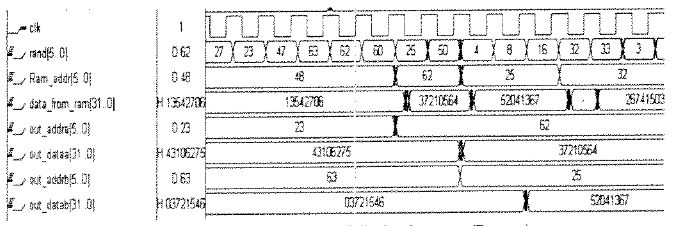


Figure IV-E: Sim. of Population of Solutions – Part A

The random numbers 62 and 25 are used as RAM addresses. A short time after address 62 is set, the chromosome 37210564 is obtained from RAM. The contents of addresses 62 and 25, along with these addresses are then set on the Module's output lines.

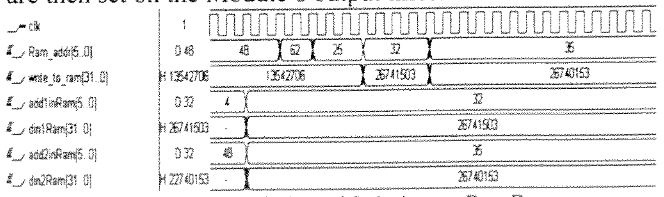


Figure IV-F: Sim of Population of Solutions – Part B

The addresses 32 and 35 and the chromosomes 26741503 and 26740153 are set to the inputs of the Population Selection Module. At a later point in time, the addresses are placed on the RAM address line, and the chromosomes are written into the RAM.

V. CROSSOVER AND MUTATION

Due to the nature of the TSP, genuine crossover cannot be performed, as it may cause duplicate cities within the tour, or the absence of certain cities. Genuine mutation cannot occur either, as flipping a random bit may introduce a city that is not in the tour. The method used for crossover and mutation is based on the original crossover and mutation algorithms, but is slightly modified in order to prevent 'nonsense' solutions from entering the chromosome population.

To produce a child solution, the following steps are followed within the crossover and mutation module.

1. Select the primary parent, and latch it to the gene_to_cross buffer. Latch also the secondary parent to the process buffer.
2. Mutate the gene_to_cross buffer.
3. Scan and latch genes that do not exist in the gene_to_cross buffer, from the process buffer.

A. Crossover

In this design, the first half of each parent chromosome is duplicated in one child. The second half of each child consists of the remaining genes in the order they appear in the second parent. Figure V-A illustrates how the genes are selected for a child from the two parents. The primary parent provides a child with the first half of its genes, while the secondary parent provides the remaining genes.

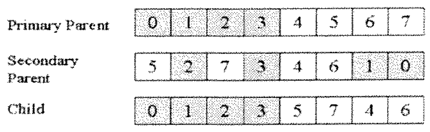


Figure V-A: Crossover selection for a child solution

The Crossover module selects which parent will be used as the primary gene provider with a switch signal line. As dynamic access to places in latched buffers cannot be achieved in FPGA programming, two internal shift-registers are used for the crossover processes; Figure V-B illustrates the crossover process adopted in this design.

At the beginning of each crossover process, a shift-register latches the secondary parent according to the switch signal, while the second shift-register latches the first half of the primary parent. The content of the first shift-register is then compared with all genes on the second shift-register in order to determine which genes are already present in the child. This is accomplished with a loop that loops for the same number of times as the number of genes. During each iteration, the shift-register shifts out one gene, which places the next gene to scan at the place where the comparison will take place. When the comparison indicates that the current gene is not from the first shift-register (first half of the primary parent), the gene is latched and appended to the second shift-register. At the end of the iteration, the second shift-register contains the first half of the primary parent and the remaining genes in the order of appearance from the secondary parent.

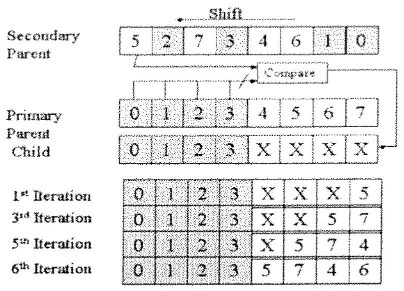


Figure V-B: Crossover process

B. Mutation

The mutation process randomly chooses a gene found in the first half of the primary parent, and places it in the last gene position in the primary parent. A three bit random factor is provided from the random number generator. This value is used to determine which gene will be mutated.

As mentioned before, two shift-registers are used in the crossover process. Mutation is implemented in a way that modifies the shift-register process, (**first register above**) before the crossovers actually takes place. The mutation logic can be best understood by examine Figure V-C shown below:

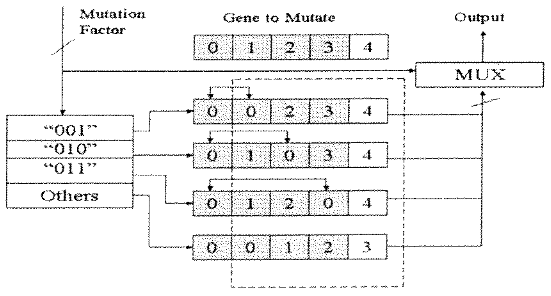


Figure V-C: Mutation logic

The Mutation logic prepares a gene_to_cross buffer that represents the mutated substitution for the first half of the parent gene. As one can see in Figure V-C, the mutation factor selects which parts of the gene_to_cross buffer will be mutated. The mutation logic replaces the gene position indicated by the mutation factor, with the first gene. The non-mutation logic simply shifts the genes by one position to the right. The genes that are used in the crossover process are the last four genes in the gene_to_cross buffer.

C. *Simulation of the Crossover and Mutation Module*

To verify the crossover and mutation method described above, the following figure shows the result of a crossover operation on both a normal crossover and crossover with mutation.

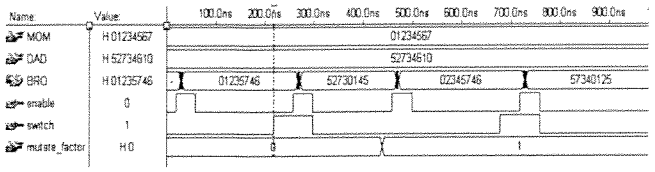


Figure V-D: Simulation of Crossover and Mut. Module

VI. FITNESS EVALUATION

The Fitness Evaluation Module receives two parent and two child chromosomes from the Crossover and Mutation Module, and determines which two chromosomes represent the shortest tour. The two chromosomes with the shortest tour are written back into the population, and their tour lengths are written to a RAM module. The contents of the RAM module are used for testing purposes. This module is enabled by the Fitness_Enable line set by the controller. The following diagram illustrates the main components of the Fitness Evaluation Module.

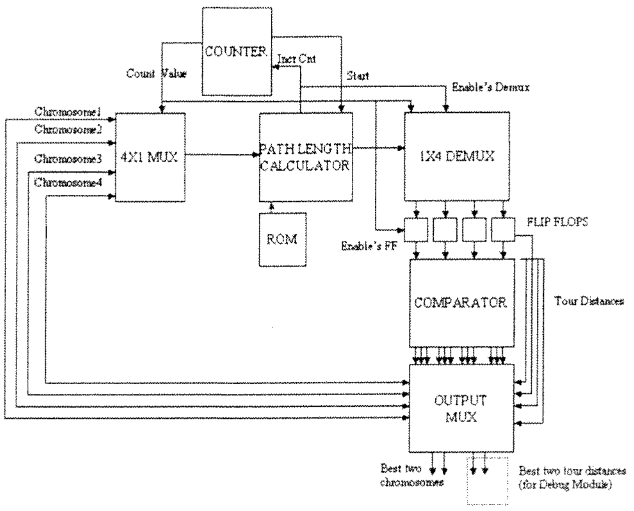


Figure VI-A: Fitness Evaluation Module

The following steps illustrate the functioning of the entire module.

1. Two parents and two children chromosomes are the input to the mulitplexer. The Counter sets the input multiplexer's select line, and sends a 'start' signal to the Path Length Calculator.
2. The Path Length Calculator accesses the ROM with each set of sequential genes in the chromosome. The genes are used to determine which address holds the distance between the two adjacent genes (cities). A running sum is kept of the total tour length. It is incremented after each ROM access.
3. Once the Path Length Calculator has calculated the total tour length, it sends an 'increment count' signal to the counter. This signal is used to enable the demultiplexer. The count value is used both as the select signal for the demultiplexer, and as an enable for a flip flop that stores the tour length.
4. The above three steps are repeated four times in total. Once for each chromosome. Once all four tour lengths have been latched into four flip flops, the values are fed into a comparator. The comparator determines the two shortest tour lengths.
5. A signal from the comparator is used to select the two chromosomes with shortest tour lengths. These chromosomes will be latched into a pipeline buffer, and later written back into the population.
6. For testing purposes, the tour lengths of the two best chromosomes are written into a RAM module. See section VIII for a more detailed description of the testing procedure.

The main components of the fitness module are the ROM Module, the Counter Module, the Path Length Calculator Module, the Comparator, and the Output Multiplexer Module.

A. *ROM Module*

A ROM table is used to store the distances between the cities. This table is represented by an 8 x 8 matrix. This allows the distances between the cities to be easily determined. The address in which the distance between two cities is stored is determined from the starting city and destination city numbers. Each cell in the matrix is an 8 bit number, varying from 0 to 256.

The ROM block is constructed using the Altera LPM modules, which makes use of the memory portion of the FPGA chip. Since the LPM ROM is formatted as a one dimensional array of words, calculation is required to map the (X ,Y) coordinates into the displacement of the memory. This is done by the function:

$$\text{Displacement} = X + 8Y$$

The ROM is initialized with a memory initialization file (.mif files). For simulation purposes, a simple .mif file generator was written in C (see Appendix). This program generates random values for the matrix. To improve access time, the ROM uses an asynchronous read which allows the ROM access to be completed within one clock cycle

B. *Counter Module*

The counter module is a 3 state Finite State Machine, based on a 3 bit up-counter. The count value is used to select which chromosome will be inputted into the Path Length Calculator Module, which demultiplexer line the tour length will be routed to, and which flip flop will be enabled to latch the tour length. The three states of the FSM include IDLE, PROCESSING, DONE. Figure VI-B illustrates the FSM.

IDLE: If the enable line has been set to '1' by the controller, then a signal telling the Path Length Calculator Module to start is set to '1'.
PROCESSING: If the increment count line is set to '1' by the Path Length Calculator Module, the count value is incremented. If the count has been incremented the next state is IDLE, otherwise the state is PROCESSING.
DONE: The count is reset to its initial value.

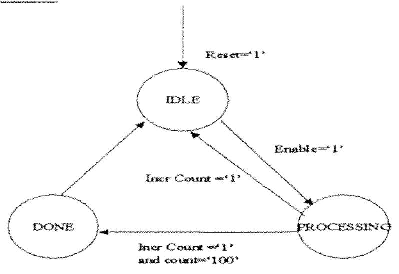


Figure VI-B: FSM for Counter Module

C. Path Length Calculator Module

This module receives a chromosome from the multiplexer, accesses the ROM module to retrieve the path lengths of each pair of adjacent cities, and outputs the path length to the demultiplexer. It also sets the signal 'increment count' to '1' when it has finished calculating a tour length. This signal tells the Counter Module when to increment its count.

The Path Length Calculator Module is a 10 state Finite State Machine, consisting of the states described below. Figure V-C below illustrates the FSM.

IDLE: In this state, the increment count line connected to the counter is set to '0', and the sum is reset to '0'. If the start signal from the counter is set to '1', then the input is ready. The most significant four bits are set to ROM addressA and the second most significant four bits are set to ROM addressB. The state then changes to SEGMENT1. If the start signal is set to '0', the ROM addresses are not set, and the state remains IDLE

SEGMENT1: The path length for the first two adjacent cities is retrieved from the ROM. The address lines are set to the second and third most significant four bits in the chromosome. The state then changes to SEGMENT2.

SEGMENT2: The running sum is set to the path length retrieved in the previous state, plus the current value of the sum. The path length is then retrieved from ROM, and the address lines are set to the third and four most significant four bits in the chromosome.

SEGMENT3 through SEGMENT8: These states function the same way as SEGMENT2. Which four bit number the address lines are set to, changes in every state. In SEGMENT8, the least significant 4 bits are set to ROM address 1 and the most significant 4 bits are set to ROM address2.

CALC_LEN: The total tour length is calculated, the 'increment count' line is set, and the state changes to IDLE.

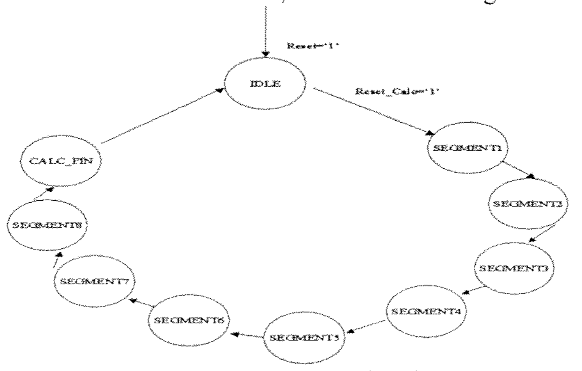


Figure VI-C: FSM of the Path Length Calculator Module

D. Comparator Module

This module consists of six Altera LPM comparator modules, which output a '1' if the first input is less than or equal to the second input. The four calculated total tour lengths are fed into the comparators. Different combinations of the six comparator output signals, and their complements are ANDED in order to determine which sum is the smallest, and which sum is the second smallest.

E. Output Multiplexer Module

This module consists of two multiplexers and a selector module. It takes as inputs, the outputs of the Comparator Module, and the four chromosome sequences. The selector module sets the select lines for the multiplexers, and outputs the best sum and second best sum. The multiplexers output the best chromosome and the second best chromosome. The chromosomes will then be latched in a pipeline buffer before being written back into the population. The sums will be written into a RAM module used for testing purposes.

F. Simulation of the Fitness Module

The following figures illustrate the functioning of the fitness module.

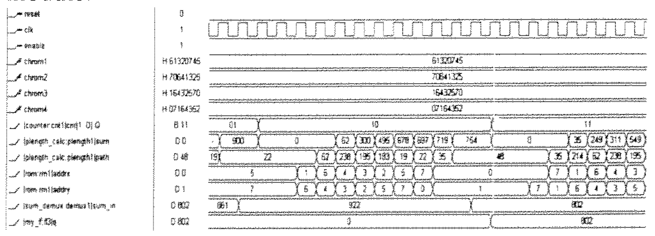


Figure VI-D: Sim. Resuts of the Fitness Module: Part A

In the above figure, when the counter value is 2, the tour length of chromosome3 (16432570) is being calculated. The ROM addresses are addrx, addry and the value accessed is the port named path. The port named sum is the running sum of all the path values. The tour length, 802 is routed through the demultiplexer, and then latched in a flip flop.

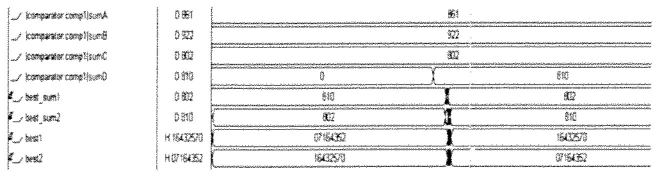

|comparator:comp1|sumA 0 861 861
|comparator:comp1|sumB 0 922 922
|comparator:comp1|sumC 0 802 802
|comparator:comp1|sumD 0 810 0 810
best_sum1 0 802 810 802
best_sum2 0 810 802 810
best1 H 16432570 07164352 16432570
best2 H 07164352 16432570 07164352

Figure VI-E: Sim. Results of the Fitness Module: Part B

In figure VI-E, the tour length of the four chromosomes, are input to the comparator. Best_sum1, and best_sum2, and best1 and best2 are the output of the Output Muliplexer module.

VII. PIPELINE OPERATION

Pipelining is integrated within the design to optimize performance. The concept of pipelining requires that the input signal to each stage of the pipeline be stable at the time the slowest stage is finished. This is achieved by employing pipeline buffers that latch the outputs from each of the stages. The latched data include: the address in RAM where the solution is fetched, and the tours. The design has three pipeline stages, therefore 3 sets of pipeline buffers exist.

The pipeline buffers are controlled by the main controller. The controller signals to the pipeline buffer when to latch new data from the previous stage. The time between latches is equal to the amount of time it takes for the slowest stage to complete. Currently the slowest stage is the fitness stage, which takes 48 clock cycles.

Due to pipelining, WAW errors could occur in this design. This error occurs when the same population address is selected by consecutive stages within the pipeline. It is possible that a better solution can be overwritten with a worse solution, if the address has not been rewritten, before it is accessed again.

To eliminate the WAW errors, a random number filter is implemented in the population selector module. The filter prevents the same number to occur at 3 consecutive stage of the pipeline.

The filter compares the current random number with the last 3 numbers generated. These numbers are stored in latches. If a duplicate number is found within the buffer, the current random number will be modified before it is latched. The modification will reduce the occurrence of consecutive duplicate numbers.

When duplicate numbers exist, the new modified number is produced by XORing two of the latched numbers. With the XOR operation, it is only possible to get an identical number when one of the numbers contains all 0's. The filter first checks if any number in the buffer is 0, when it sees zero, it XOR's a hardwired number with one of the numbers in buffer, otherwise it XOR's both of the buffered numbers to produce the current number. As a result, the new number is completely different from the numbers in the buffer. This reduces the occurrence of WAW errors, which can impede the convergence of the population. The design flow is shown below in Figure VII-A.

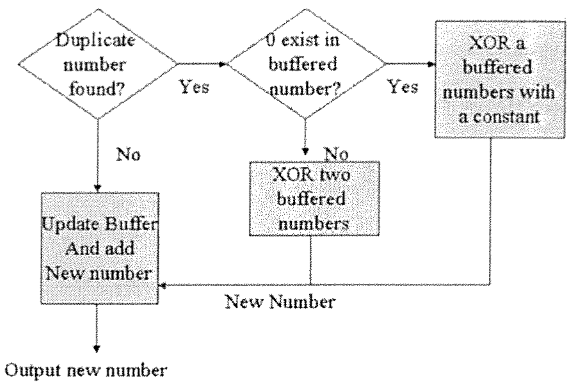

Duplicate number found?
Yes
0 exist in buffered number?
Yes
XOR a buffered numbers with a constant
No
No
XOR two buffered numbers
Update Buffer And add New number
New Number
Output new number

Figure VII-A: Flow chart for Random number filter

VIII. ANALYSIS

The overall design uses 1970 logic cells of the FPGA. This corresponds to 85% of FLEX10K40RC208-3 chip. The memory component utilizes 3264 bits, which is 19% of the device's memory. This relatively small resource usage leaves space for implementing more complex version of the GA modules.

A. Maximum Frequency Analysis

Each of the modules within this design has independent timing requirements. The longest path from the inputs to the outputs, for each stage, is determined by performing several timing analyses. The results are shown in table VIII-A below:

Population selector	Crossover & Mutation	Fitness testing
15.6ns	12.8ns	40.9ns

Table VIII-A: The longest path for each pipeline stage

Table VIII-A indicates that the Fitness Module requires 40.9 ns to produce stable output, thus the **maximum frequency** of the design is 1/40.9ns = **24.4Mhz**

B. Performance Analysis

The clock cycles needed for each pipeline stages can be obtained from the number of states in the finite state machine in each pipeline stage. The results are shown in Table VIII-B.

Population selector	Crossover and Mutation	Fitness testing
15 cycles	6 cycles	48 cycles

Table VIII-B: Clock cycle required for pipeline stages

As shown in Table VIII-b, the non-pipelined implementation would require a minimum of 15 + 6 + 48 = 69 clock cycles per iteration of the genetic algorithm, which is approximately 2.83μs. In contrast, the pipelined implementation requires 48 clock cycles for each pipeline stage, therefore each iteration of the genetic algorithm would take approximately 48 clock cycles, which gives 48 * 41ns ≈ 2μs per iteration.

Therefore, the overall improvement for employing the pipeline technique is:

$$\frac{2.83\mu s - 2\mu s}{2\mu s} * 100\% = 41.5\%$$

Pipelining yields a **41.5%** improvement in speed.

As the reader might have notice, the Fitness testing stage is the performance bottleneck of the design, however this module was designed with space minimization in mind.

To evaluate the solutions obtained from the genetic algorithm, and to determine whether the solution converges, a Debug RAM module is implemented. This module stores the tour length of the fittest solutions each time two chromosomes are written back into the population. While a simulation of the design is performed, the contents of the Debug RAM are observed. In order to determine whether the solution converges, a simulation was run over 1.5ms (this corresponds to 37 iterations of the GA). The tour length values tend to decrease over time. This indicates the worse solutions are weeded out. Figure VIII-A, shows the average tour length over time, for a population of 64 chromosomes.

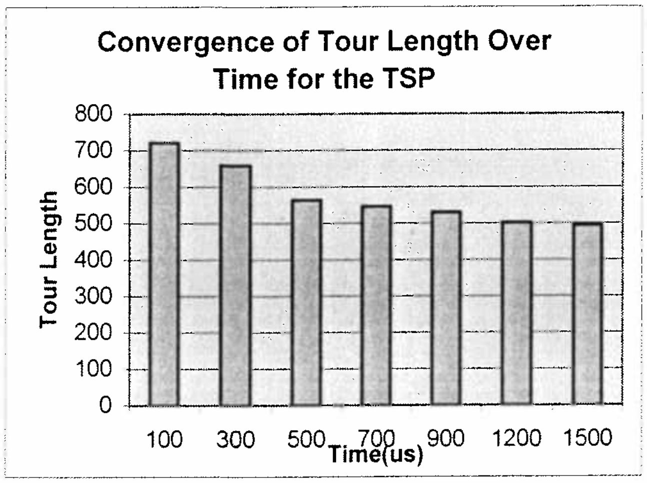


Figure VIII-A: Convergence of Tour Length Over Time

IX. FUTURE WORK

There are many aspects of this design that could be improved, or modified. The speed of the fitness module could be improved, if space is compromised. Also, introducing a crossover probability would allow the design to be more consistent with a pure GA, and could lead to a greater rate of solution convergence. A different crossover algorithm could be implemented, such as a stochastic method, and the performance of the two algorithms could be evaluated. Tournament selection or Rank selection, are two chromosome selection methods that may improve the performance of the design. Had there been more time, these alternative methods would have been investigated.

X. CONCLUSIONS

The goal of this project was to design route optimization hardware using a Genetic Algorithm, and show that this implementation provides a solution to the Traveling Salesperson Problem. Our design maps well into an Altera Flex10K FPGA, as it uses only 1970 logic cells, and 3264 bits of memory. Also, it is fast, as it can operate at a maximum frequency of 24.4MHz. Our simulation results indicate that the solution to the TSP converges over time, thus our design meets all project expectations. In addition to the speed advantage a hardware implementation of a GA could have over a software implementation, the applications of a hardware implementation are plentiful. Work done by Tommiska & Vuori [1] indicate hardware GA's may be particularly suited to the optimization of routing in ATM networks. Our design could easily be applied to applications such as this one.

XI. REFERENCES

[1] M. Tommiska and J. Vuori. Implementation of genetic algorithms with programmable logic devices. In J.T. Alander, editor, Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA), pages 71-78, August 1996.

[2] Thomas Dean, James Allen and Yiannis Aloimonos. Artificial Intelligence Theory and Practice. Addison-Wesley Publishing Company, New York, 1995.

[3] http://cs.felk.cvut.cz/~xobitko/ga/

A VHDL Implementation of an IEEE-754 Standard Single Precision Floating Point Unit

Jeffery Montesano, Nicholas Chan and Eric Chung Kam Chung

Abstract—Floating point is non-integer representation of a number; the number is represented as significand and exponent. Hardware that performs arithmetic operations using this representation is known as a floating-point unit (FPU). FPUs can be found in virtually every computer today, either integrated within a CPU (such as Pentium or PowerPC) or as a separate specialized unit (such as graphics accelerators). They allow calculations to be performed with a high degree of precision and speed.

This paper presents an implementation of an FPU in VHDL complying with the IEEE-754 standard. The IEEE-754 standard specifies the actual representation of the number and the handling of special cases such as "NaN" (not a number), "infinity" and rounding schemes. The FPU is able to perform the basic single-precision operations of multiplication, addition, subtraction and division. The IEEE-754 single precision number is based on a 32-bit word: 1 bit for the sign, 8 for the exponent and 23 for the mantissa.

The basic architecture of the FPU consists of a centralized controller and three separate functional sub-units. The division of arithmetic operations into three separate functional units allows the FPU to implement instruction level parallelism (ILP).

Keywords—algorithms, floating-point, ILP, rounding.

I. INTRODUCTION

FLOATING-POINT arithmetic was first implemented in the computers of the late 1940's. However, the decision to include floating-point arithmetic in the early computers created a controversy. The computers of the day only implemented exact integer arithmetic. It was argued that implementing floating-point arithmetic would be a waste of (the then) precious memory since the computations using real numbers could be done in software using *floating vectors* and *scaling factors* and integer operations. The drawbacks of using this scheme quickly became apparent. Implementing floating point arithmetic became widespread by the late 1950's [2].

By the early to mid 1980's, companies such as Intel, Motorola and Macintosh were implementing their own FPUs and including them in computers that were accessible to the general public. However, no standard on the exact representation of the floating-point had been decided upon. The final standard, IEEE-754, was made official only in 1985.

A. The IEEE-754 Standard

The IEEE-754 standard for single precision floating-point numbers has the following format:

$$(\text{sign})1.\text{mantissa}*2^{\text{exponent}-127} \qquad (1)$$

It provides 1 bit for the sign, 8 bits for the exponent and 23 bits for the mantissa for a total of 32 bits. To allow for more precision in the mantissa, the IEEE-754 standard makes the leading '1' in front of the decimal implicit. Also, the exponent is stored in a bias 127 format to avoid the use of a two's complement representation. In this case, -2 would be represented as $-2+127_{\text{ten}} = 125_{\text{ten}} = 01111101_{\text{two}}$. Using this scheme, a range of abs($1.175*10^{-38}$, $3.4*10^{+38}$) can be represented. The precision that can be achieved is:

$$\begin{aligned}\#\text{ of decimals points precision} &= (23 \log 2)/(\log 10) \\ &= 6.9 \qquad (2)\end{aligned}$$

Results using this representation are accurate up to the sixth decimal point.

The IEEE-754 standard provides four rounding modes as shown in Table I.

TABLE I
IEEE-754 Rounding Modes

Rounding Mode	Sign of Result ≥0	Sign of Result < 0
–		+1 if r OR s
–	+1 if r OR s	
0		
Nearest	+1if (r AND po) OR (r AND s)	+1if (r AND po) OR (r AND s)

The symbols *r* and *s* represent the *round* and *sticky* bits, which are obtained during computation of an answer. They are used to make the final decision on whether to round or not depending on the mode. A 3rd bit, *g* or *guard* bit, is also used in the rounding process. Its purpose is to update *r* and *s* during computation.

Table II shows the IEEE-754 encoding to represent values such as $+\infty$, $-\infty$, *NaN*.

Manuscript received December 3, 1999.
Jeffery Montesano (9728164) is a student of Computer Engineering at Mcgill University, Montreal, Quebec.
Nicholas Chan (9728058) is a student of Computer Engineering at Mcgill University, Montreal, Quebec.
Eric Chung Kam Chung (9729176) is a student of Computer Engineering at Mcgill University, Montreal,Quebec.

TABLE II
IEEE-754 Representation of Special Values

Exponent	Mantissa	Represents
e = Emin − 1	f = 0	±0
e = Emin − 1	f ≠ 0	$0.f{*}2^{Emin}$
Emin ≤ e ≤Emax	–	$1.f{*}2^{Emin}$
E = Emax + 1	f = 0	±∞
E = Emax + 1	f ≠ 0	NaN

B. Design Objectives

Today, FPUs are found in virtually every computer system and used in virtually every application: scientific, business and even leisure. With computing power doubling every 18 months, floating-point unit architectures are being developed to take advantage of this fact. One method is known as *Instruction Level Parallelism* or ILP. ILP is a scheme in which more than one instruction can be executed simultaneously giving rise to an increased performance. The main goal of this project was to implement, in VHDL, an FPU capable of multiplication, division, addition and subtraction with an ILP of three instructions.

II. Design Methodology

A top-down design approach was taken for this project. The required tasks of an FPU when interfaced with a RISC computer architecture were examined. It was determined that the FPU must:

1. Receive instructions from the instruction register.
2. Compute the result.
3. Write the result to a file register.

Next, the overall architecture of the FPU was determined by examining possible methods of implementing ILP. The obvious choice would have been the use of pipeline registers at every stage of the computation process. However, due to lack of time, an alternate method was used.

The alternate method separated the arithmetic operations into three functional sub-units: one for multiplication, one division and one for addition/subtraction. Since each sub-unit is independent of each other, simultaneous computations are possible. However, to provide functionality, a main controller is necessary to maintain proper data and control flow between the functional sub-units and act as an interface to the CPU. The architecture of the FPU can be seen in figure 1.

The FPU must now:

1. Receive instructions from the instruction register.
2. Route the operands to the appropriate functional sub-unit based on the *opcode*.
3. Compute the result.
4. Write the result to a file register.

It is assumed that each functional unit has its own write port on the register file

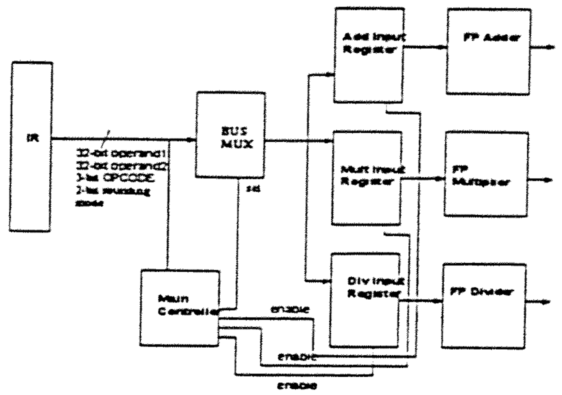


Figure 1: Architecture of Floating Point Unit

III. Design Implementation

The actual implementation of the floating-point unit was done in VHDL using Altera's MAX+PlusII software. The target technology was Altera's Flex10K family.

A. Main Controller

The architecture shown in fig.1 suggests a relatively simple main controller, consisting entirely of combinational logic. Its function is to receive the opcode from the recently latched instruction in the instruction register, and enable the input register of the appropriate functional sub-unit based upon the opcode received. While this may seem a little simplistic, more thought was required than this. Firstly, the functional sub-units are all unpipelined and therefore cannot handle overlapped instructions. As a result, the main controller cannot be allowed to send an instruction to a functional unit while it is busy. This leads to a handshaking protocol, whereby the controller is only allowed to enable the input register of the target functional unit provided that unit is not busy. The busy signal comes from the dedicated controller of each functional sub-unit. It is asserted during computation and deasserted otherwise. This eliminates the possibility of the input register ever being written during computation, which would unquestionably lead to calculation error. The routing of the instruction is accomplished by a "busmux", which takes as input the 69-bit instruction (two 32-bit operands, a 3-bit opcode and a 2-bit rounding mode) and uses the opcode to determine where to route it. The enable line is not only sent to the input registers, but also to the dedicated control units of each functional unit, giving the "start" signal to begin computation.

The input registers of the functional units are interesting in themselves, as they perform a very important task required by the IEEE-754 standard. Namely, they "unpack" the mantissa making the implicit '1' explicit. This is necessary since all of the arithmetic algorithms use this value to compute their results. As a consequence, the input registers accept a 23-bit mantissa, but output a 24-bit mantissa for use by the functional unit with the most significant bit hardwired to '1'. The input registers are also accompanied by multiplexors, allowing them to "load" the next value, or "remember" the last value. This is illustrated in figure 2.

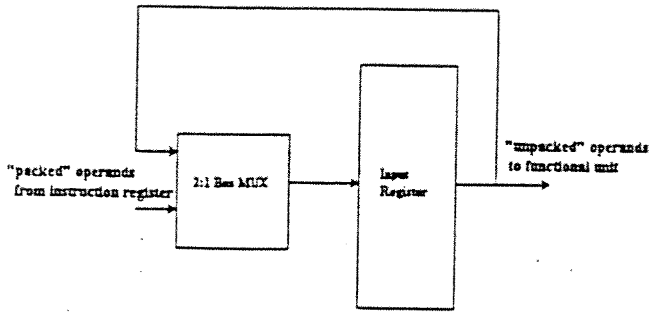

"packed" operands from instruction register
2:1 Bus MUX
Input Register
"unpacked" operands to functional unit

Figure 2: Functional Unit Input Register

B. Division

Floating point division is very similar to integer division algorithmically. It essentially involves subtracting the exponent of the divisor from that of the dividend, and performing integer division on their unpacked mantissas.

Of the many binary division algorithms, a decision was made to use the "restoring division" algorithm based on its simplicity. It is basically equivalent to the paper and pencil method used in base-ten arithmetic: find the largest multiple of the divisor that can be subtracted from the dividend, creating a digit of the quotient on each attempt. Binary division involves only two numbers ('0' and '1') thereby simplifying the algorithm: the multiple of the divisor either goes into the dividend zero times or one time. To perform the restoring division algorithm the *n-bit* divisor is stored in an *n-bit* register, while the *n-bit* dividend is stored in the right half of a *2n-bit* left-shift register. The divisor is subtracted from the dividend, and if the result is positive then a '1' is shifted into the quotient register, otherwise the dividend is "restored" and a '0' is shifter into the quotient. This may seem puzzling at first glance, since the divisor is remaining static (not increasing in size), while the dividend is shifting left (increasing in size). The grammar school algorithm prescribes exactly the opposite i.e. keep the dividend static and shift the divisor right (having started in the left half of a *2n-bit* register), making the divisor decrease in size until the largest possible multiple of it can be subtracted from the dividend. As it turns out, computer architects discovered long ago that making the dividend shift left yields the same result as making the divisor shift right [2].

Dividing two unpacked 24-bit mantissas requires altering the division algorithm slightly, since the numbers will always be aligned (1.mantissa$_{\text{dividend}}$ / 1.mantissa$_{\text{divisor}}$). This means that there is no need to start the dividend in the right half of a *2n-bit* register, or any need for a *2n-bit* register at all. The dividend can be stored in an *n-bit* shift-left register, the divisor in a static *n-bit* register, and the quotient in a shift-right *n-bit* register. The hardware used to implement the mantissa division is shown in figure 4 [1].

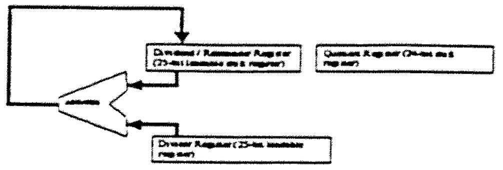

Dividend / Remainder Register (23-bit mantissa shift register)
Quotient Register (24-bit shift register)
Divisor Register (23-bit mantissa register)

Figure 4: Hardware used to implement mantissa division

After performing the algorithm for twenty-six iterations (two extra iterations to calculate round bits), the most significant bit (MSB) of the result is examined: it could be either a '1' (meaning the result is properly normalized as 1.mantissa) or a '0' (meaning that normalization is necessary). If the MSB is a '0', then the 2nd MSB ,*must* be a '1', simply due to the fact that the operands were normalized prior to calculation. This situation requires a shift left of the mantissa, and a subtraction of one from the exponent.

Computing the result's exponent seems straightforward at first glance: simply subtract one exponent from the other, and store the result. Unfortunately it is not that simple. First off, the exponents come into the divider as 8-bit unsigned bias-127 binary numbers. Just subtracting one from the other would result in an answer with a bias of zero:

Consider the following situation:
Dividend exponent = (5 + 127) = 132
Divisor exponent = (2 + 127) = 129

Subtracting the two exponents would result in a value of 3, which corresponds to (3 – 127) = -124 in IEEE-754 notation. Clearly, the bias must be added to the result in order to conform to the standard. Thus, 3 + 127 = 129 is what would need to be stored in the result's exponent register.

Another obstacle arises from the problem of exponent overflow and underflow. If the exponents are subtracted from each other (with the bias added), and the result is zero or negative, then underflow has occurred (the exponent is less than 2^{-126}). On the other hand, if the exponents are subtracted and the bias added and the result is larger than 254, then overflow has occurred (the exponent is larger than 2^{127}). To properly detect both overflow *and* underflow, a 10-bit representation of the 8-bit exponent is needed. This works as follows: upon entry to the functional unit, the exponent is "padded" with two 0's at the two MSB's. The subtraction occurs on the two-padded exponent, the bias is added, and then the two MSB's of the result are examined. A '1' in the most significant bit of the 10-bit result indicates that the result is negative (since addition and subtraction are performed in two's complement), and therefore underflow has occurred. A special check is also made to see if the resulting exponent is zero, since this is a reserved value, and results in underflow as well. A '1' in the second MSB means that the result is larger than 254, and therefore overflow has occurred. Again, a special check is made to see if the result is 255, which is a reserved value, and also results in overflow.

Taking care of the sign is a simple matter in floating point division, simply taking the XOR of the signs of the operands and storing them in the output register.

Having explained the basics of floating point division, it is now time to look at special cases. Division offers more chance for an invalid operation than any other arithmetic operation, requiring painstaking exception checks every step of the way [1]. Upon entry to the division unit, the operands are passed to an exception check unit which checks for every possible invalid case. This includes operations such as zero divided by zero resulting the in special value NaN, a number divided by zero resulting in the special value ∞, and many other cases. When a special case is encountered at this early stage in the computation the entire data path is bypassed, and the control unit writes the result to the output register. This type of operation requires the use of multiplexors at the output register controlled by the control unit, selecting whether the result should come from the data path or from the control unit.

The rounding was implemented according to table I, with an exception check performed afterwards for overflow. An interesting case to be covered is the case where the largest possible mantissa (1.1...1) is divided by the smallest possible mantissa (1.0...0) resulting in a mantissa of 1.1...1. If the result's exponent is 127 (254 in IEEE notation) (the maximum possible) and a round-up occurs, then the divider will add one to the mantissa, causing it to reset to 0.0...0, and the exponent is incremented accordingly. At the exception check stage, overflow will be detected since the exponent has bypassed the allowable range.

C. Multiplication

The floating-point multiplier was designed the same way as the other floating point units discussed. It contains a finite state machine controller that controls the data path. The algorithm for floating point multiplication is investigated next.

$$\text{multiplicand} = mA * 2^{eA}$$

$$\text{multiplier} = mB * 2^{eB}$$

where mA and mB are each unpacked 24 bit mantissas. Multiplying the two numbers results in the following product:

$$(mA * 2^{eA}) * (mB * 2^{eB}) = mC * 2^{eC} \quad (3)$$

where $mC = (mA * mB)$

$eC = eA + eB - \text{bias } 127.$

Notice that because the exponents are stored with a bias 127, the same bias has to be subtracted from the sum of the exponents in order to get the new biased exponent.

The algorithm for normalized floating-point multiplication illustrated in Figure 5, is a direct implementation of (3).

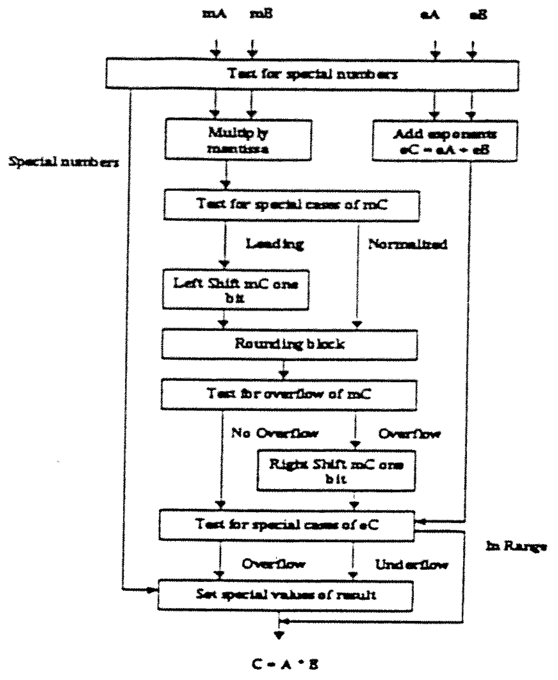


Figure 5: Floating-point Multiplication algorithm

As seen in the floating-point division algorithm, special cases must be checked before, during and after computation to make sure that all such cases are handled. First off, if the operands are special numbers, the whole algorithm is bypassed and the result is set. In multiplication some of those cases are shown in table III.

TABLE III
EXAMPLES OF SPECIAL CASES
OF FLOATING POINT MULTIPLICATION

NaN * # = NaN
0 * ∞ = NaN
0 * # = 0
+∞ * # = ∞
-∞ * ∞ = -∞

The first step after the initial check for special conditions multiplies the mantissas using ordinary integer unsigned multiplication and computes the new biased exponent. Altera LPM modules were used for both the multiplication and addition. For the resulting mC (48 bits), there are two special cases. If the result's most significant bit is in the second leftmost bit (product contains a leading zero), then a left shift of mC is necessary to normalize the result. Otherwise, C is already in normalized form, and there is a branch to the rounding stage. The rounding module implements the IEEE rounding scheme which rounds according to the round bit, guard bit and sticky bits in the least significant 24 bits of the mantissa product. In fact, any desired rounding scheme could be implemented in the rounding unit due to the modularity of the design.

After this rounding, it is possible that mC will overflow. In this case, it is necessary to shift mC right one bit and increment the exponent by one. The next step checks for special cases of eC. If there is an overflow or underflow of eC, it is handled in the "set special values of result" module. Otherwise, the result is in range, and the calculation is complete.

Just as in the floating-point division, the exponent was extended from 8 bits unsigned to 10 bits signed prior to computation to allow checking for overflow and underflow of the result.

D. Addition and Subtraction

The decision to combine addition and subtraction in the same unit was based on the algorithm chosen. The algorithm can compute sums with negative numbers. This makes subtraction a simple matter of inverting the sign of one of the operands and computing the sum. A simplified version of the algorithm is as follows:

1. Check the exponents.
2. If the signs differ, convert the 2nd operand to its two's complement.
3. Shift the smaller number right until the exponents match. Store g, r and s bits.
4. Compute the preliminary mantissa. If the answer is negative, replace it with its two's complement.
5. Shift the preliminary mantissa until it is normalized. Adjust the exponent accordingly.
6. Update r and s bits.
7. Round.
8. Compute sign.

The algorithm was first encoded into a controller as a finite state machine. To keep control complexity to a minimum level, a direct approach was taken. Every operation necessary in each step was mapped directly into hardware. For example, in steps 2 and 4, it may be necessary to perform a two's complement. Instead of having one unit perform the operation for both steps and having the inputs and outputs multiplexed, two separate units were used. This gave the data path a unidirectional flow simplifying the task of the controller; only components involved in the current step of the algorithm have to be monitored. The drawback to this approach is that more hardware resources are required. The resulting architecture is shown in figure 6. Note the multiplexors just before the final register. This allows the source of the final answer to be selected; either the computing hardware or the controller in the case of special values or exceptions. As with the previous operations, rounding is based on table I.

The stages in which underflow or overflow may occur are during the normalization and rounding process since during these stages, if the mantissa is shifted left the exponent is decremented, and incremented if shifted right. The *exponent logic* unit performs all exponent computations and monitors for underflow and overflow.

As mentioned, since there are only a finite number of bits that are used to represent mantissa, only a finite precision is possible. In the case of the adder/subtractor, the largest difference possible in the exponent is 253. However, the mantissa is represented with 23 bits. Any difference greater than 23 in the exponents of the operands, will result in the answer being equal to the larger operand.

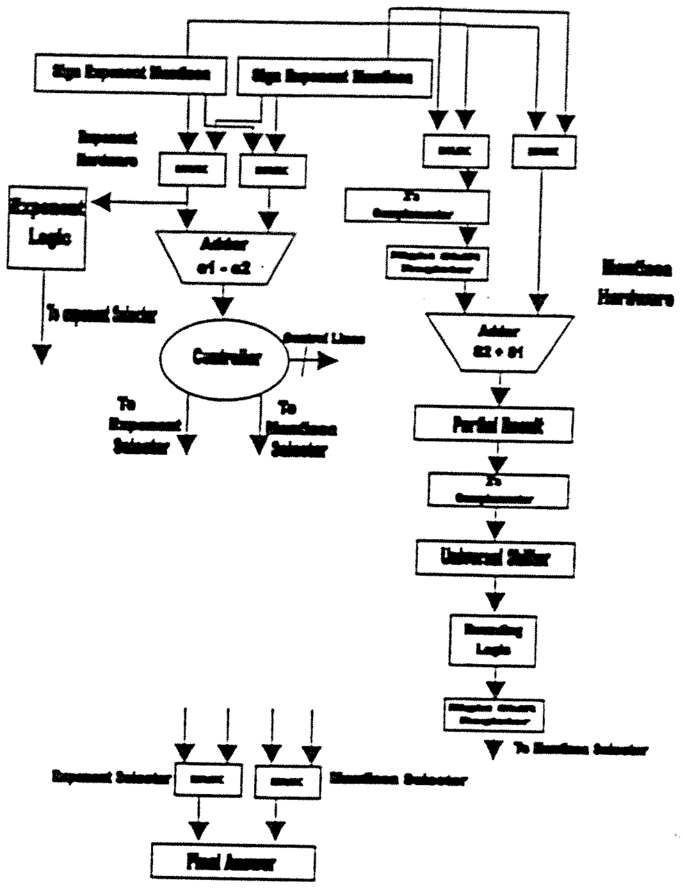


Figure 6: Architecture of Floating Point Adder/Subtractor

IV. Results and Analysis

A brief look at the results of each functional unit will provide some insight into the strengths and weaknesses of the overall design.

A. Divider Results and Analysis

The floating-point divider is capable of computing a single instruction every 74 clock cycles, most of which are spent performing the iterative restoring division algorithm on the mantissas. While very area efficient at just 510 logic cells of the Altera FLEX10K FPGA chip, the design can only be clocked at 10.12MHz. The main reason for this poor registered performance is the long path between the output of the divisor register and the input to the dividend register when performing mantissa division. This path includes a 25-bit adder and a 2:1 multiplexor. Despite the lack of performance, the divider excels in computation correctness, handling all possible cases and outputting results to six decimal places.

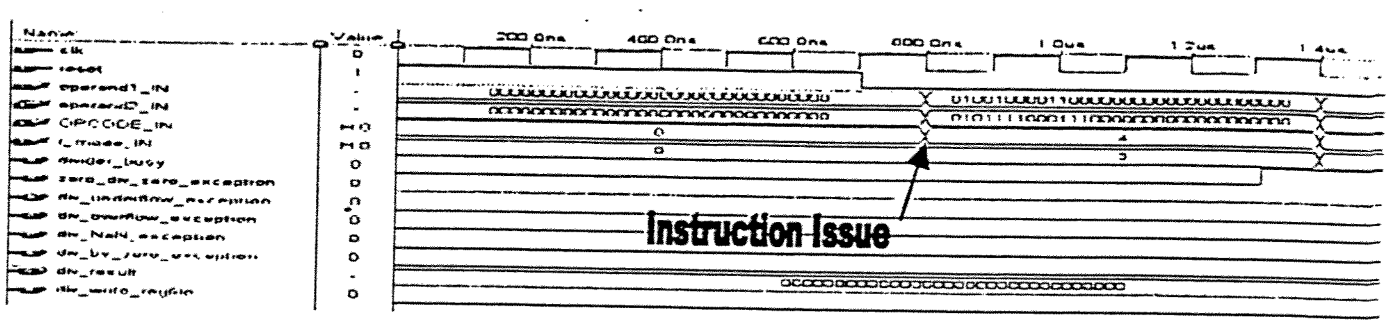


Figure 7: Simulation of Division: Issue of Instruction

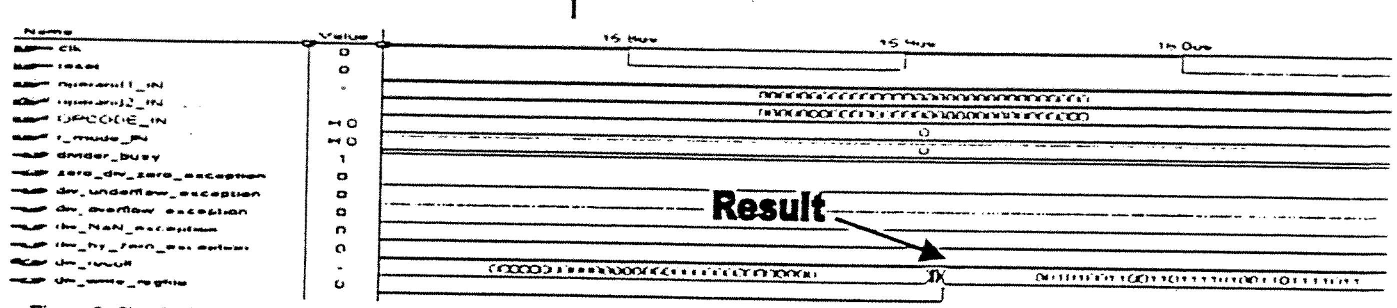


Figure 8: Simulation of Division: Result of Instruction

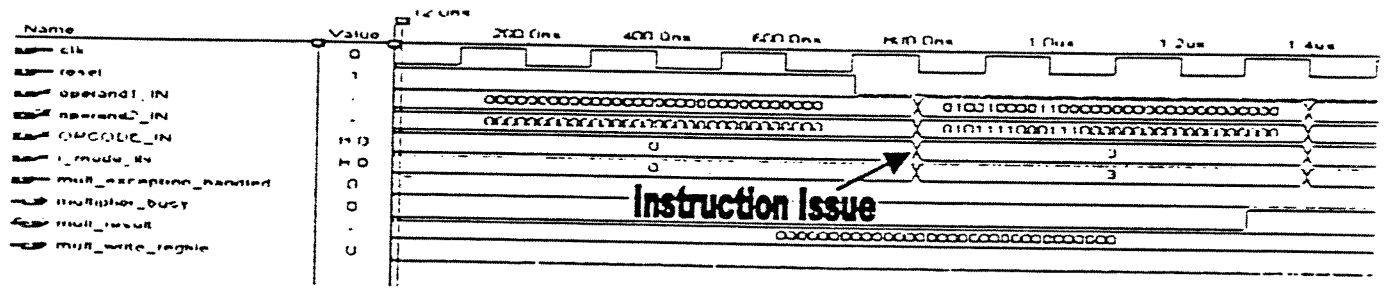


Figure 9: Simulation of Multiplication: Issue of Instruction

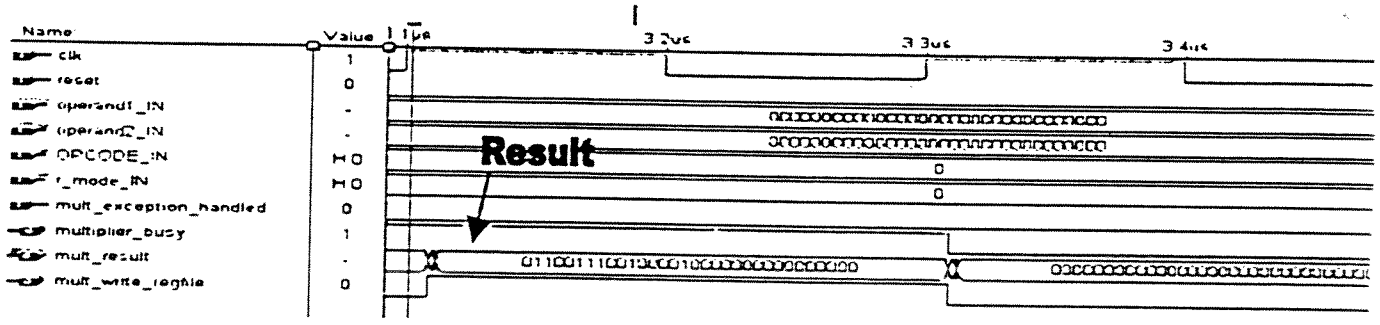


Figure 10: Simulation of Multiplication: Result of Instruction

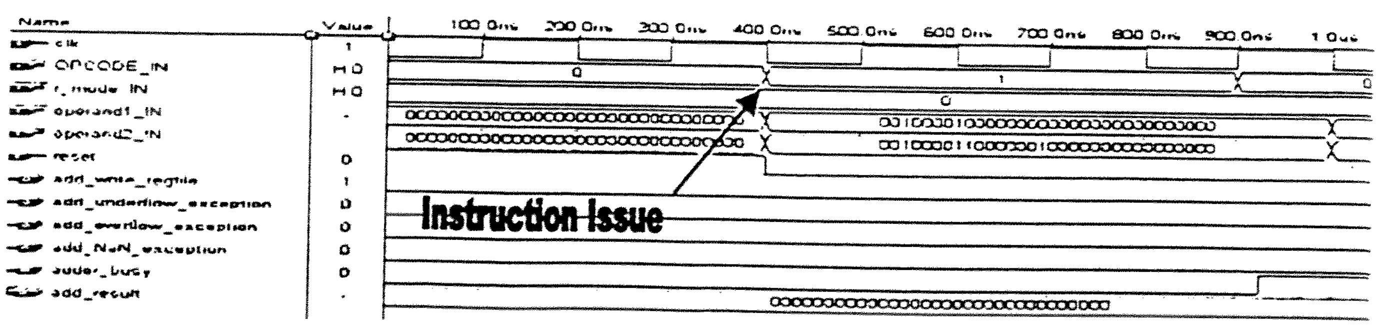


Figure 11: Simulation of Adder/Subtractor: Issue of Instruction

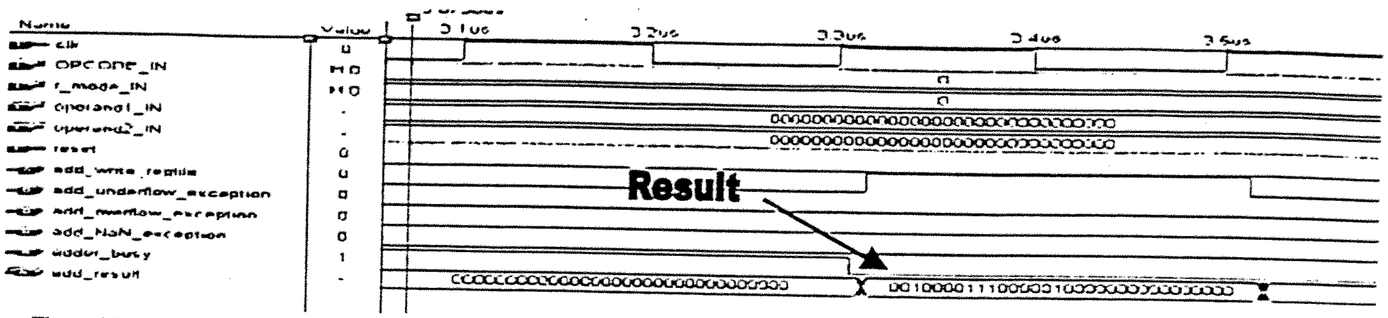


Figure 12: Simulation of Adder/Subtractor: Result of Instruction:

Figures 7 and 8 show a sample calculation performed by the division unit, split into two steps - instruction issue, and result:

The calculation performed in figures 7 and 8 is:

$$1.75\times2^{17}\div1.4375\times2^{61} \qquad (4)$$

A standard scientific calculator gives the result of (4):
$6.920068383 \times 10^{-14}$
The FPU division unit gives the result of (4):
$6.920068089 \times 10^{-14}$

B. Multiplier Results and Analysis

By far the fastest of the three functional units, the multiplier unit performs its calculation in just 11 clock cycles. Despite this small latency, the unit can only be clocked at a maximum rate of 8.95MHz. This is due in large part to the use of the Altera LPM multiplier unit, which was instantiated in its unpipelined version. This requires the multiplication of two 24-bit numbers in a single clock cycle, thereby decreasing the maximum clock rate substantially. A total of 2355 logic cells were used, making the multiplier the largest functional unit in the FPU. Figures 9 and 10 show a sample calculation using the same operands used earlier in the division example:

$$\left(1.75\times2^{17}\right)\times\left(1.4375\times2^{61}\right) \qquad (5)$$

A standard scientific calculator gives the result of (5):
$7.603010038 \times 10^{23}$
The FPU multiplication unit gives the result of (5):
$7.603010037 \times 10^{23}$
This calculation appears to have eight decimals of precision, but this is just due to rounding by the multiplier unit.

C. Adder/Subtractor Results and Analysis

The latency can vary from as little as 2 clock cycles to as much as 280 clock cycles. The shift registers used in the unit can only shift 1 bit per clock cycle. As mentioned before, the difference in the exponents can be up to 253, requiring the mantissa of the 2nd operand to be shifted 253 times. According to the registered performance function in Altera's Max+PlusII, the adder/subtractor can only be clocked at a maximum 6.93MHz. This can be attributed to the use of unpipelined Altera LPM adder units throughout the design. The adder unit required 1096 logic cells to synthesize onto the Altera FLEX10k FGPA. Figures 11 and 12 show a sample calculation using the following operands:

$$\left(1.00\times2^{-61}\right)+\left(1.0078125\times2^{-60}\right) \qquad (6)$$

A standard scientific calculator gives the result of (6):
$1.307818871 \times 10^{-18}$
The FPU adder/subtractor unit gives the result of (6):
$1.307818871 \times 10^{-18}$

The adder/subtractor unit produced the exact result. This is because the operands' exponents do not differ by a large amount. The greater the difference, the more numerous the number of shifts required leading to greater inaccuracies.

D. Integrated FPU Results and Analysis

Integration of the three functional units into a single unit was a seamless operation, as much thought had gone into the interaction between the main controller and sub-unit controllers during the design process. Since the integrated FPU incorporates three functional units of differing latencies, it makes little sense to talk of the latency of the FPU in clock cycles. The maximum clock rate for the FPU is 3.15MHz, which is slower than that of any of the three functional units. Obviously the interfacing with the main controller is responsible for this. The FPU takes a total of 4315 logic cells, occupying 87% of the Altera FLEX10K100 FPGA chip. The FPU accomplishes the task of ILP as shown in figures 13 and 14, where three instructions are issued on consecutive clock cycles and are computed simultaneously.

V. Conclusion

In conclusion, our team accomplished the goal of implementing a single precision IEEE-754 standard floating-point unit in the IEEE standard hardware description language VHDL. As the final integrated design shows, performance in terms of clock rate is not very good, but since this was not a design goal little time was spent trying to optimize it. The main design goal was to accomplish ILP and it was achieved. For increased ILP support, multiple functional units could easily be incorporated into the FPU, increasing overall throughput and performance. The final design is generic enough to fit into most RISC-type architectures, receiving instructions from a pipeline register, and writing results to a multi-port register file.

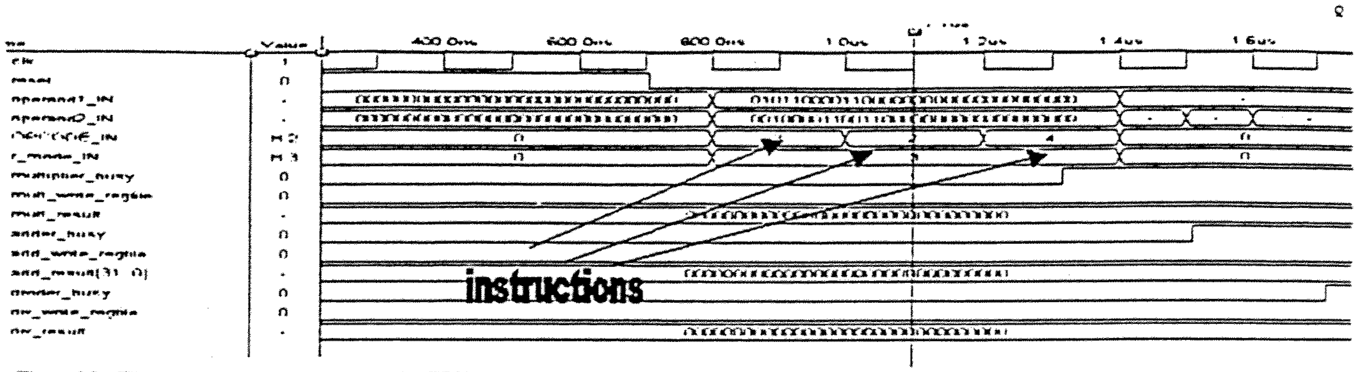


Figure 13: Three instructions are sent to the FPU on consecutive clock cycles. The busy lines react two clock cycles later.

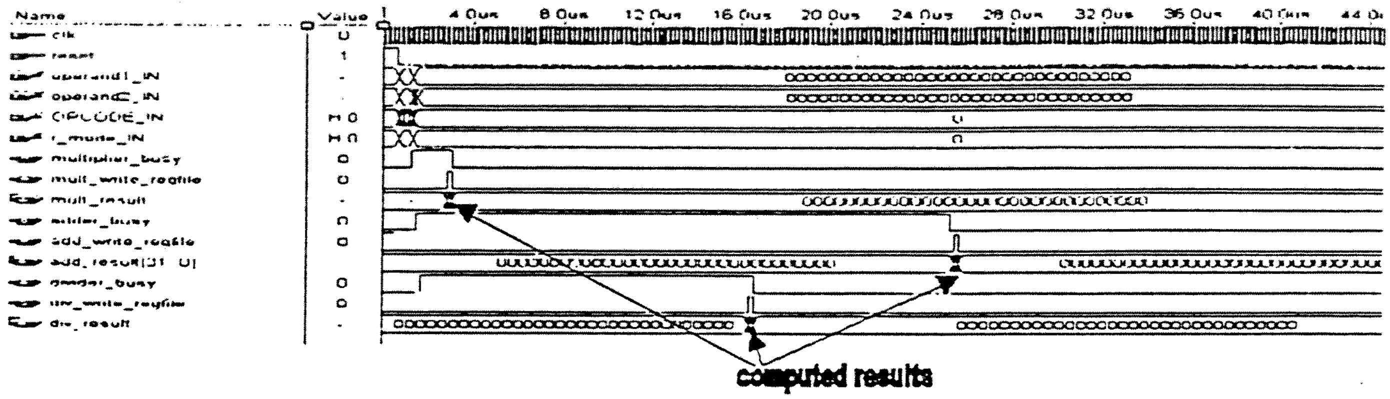


Figure 14: The answers of the three instructions are output

VI. Acknowledgment

We would like to thank the McGill University Software Engineering Lab and its staff for providing the facilities and service required to implement our project. Network administrators were available to allocate additional virtual memory required by Altera for compilation at all hours of the night.

VII. References

[1] D.A. Patterson, J.L. Hennessy, *Computer Architecture A Quantitative Approach*. San Francisco: Morgan Kaufmann Publishers, Inc., 1996
[2] D.A. Patterson, J.L. Hennessy, *Computer Organization and Design The Hardware/Software Interface*. San Francisco: Morgan Kaufmann Publishers, Inc., 1994
[3] P.J. Ashenden *The Designer's Guide to VHDL*. San Francisco: Morgan Kaufmann Publishers, Inc., 1996