FreeFood: A 1GHz High-Performance 16-bit DSP with SIMD Multimedia Extensions

Gautam Bhatnagher, Brent Climans, Andrea Pellegrini, Bo Xiao, Dan Zhang
University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109
{gautamb, climansb, apellegr, boxiao, danz} @ umich.edu

ABSTRACT
In this paper we propose FreeFood, a novel 16-bit DSP designed for video decoding and other multimedia operations. FreeFood goes above and beyond the baseline implementation in order to reach the throughput required for H.264 decoding and other complex multimedia applications. Features include a dedicated multiply-accumulate unit, 64-bit SIMD unit, and stream buffers. Many other complex and novel pipelining schemes are implemented in order to hit our targeted 1.05GHz clock speed. FreeFood can sustain a maximum throughput of 4200 MIPS while consuming only 348mW with 1.2v on the standard single-Vt IBM 130nm process. FreeFood's main target is to provide uncompromising performance while maintaining reasonable power efficiency.

Keywords
High performance, DSP, Video Encoding, H264, Multimedia.

1. INTRODUCTION
The objective of this project is to design FreeFood, a 16-bit RISC microprocessor in IBM 0.13 micron technology that can obtain a clock speed of 1GHz after accounting for clock skew. Besides implementing the baseline instructions, we support 4-way SIMD execution through multiple custom datapaths, as well as multimedia-friendly instructions such as pipelined multiply-accumulate, vector load and store, permute, and others. Due to the unacceptably slow memory units provided by Memory Compiler, we decouple the pipeline with stream buffers in order to reach our clock speed goal. The processor is composed of the custom scalar datapath, custom vector datapath, and synthesized controller module. FreeFood is targeted towards digital signal processing for embedded applications, specifically for H.264 video decoding.

2. MOTIVATION
The goal of our project is to produce a useful microprocessor with a reasonable implementation specialized for high performance H.264 video decoding. In order to meet this high performance requirement, we wish to execute up to 4 instructions per cycle through a SIMD (Single Instruction Multiple Data) implementation, as well as a high clock rate of 1GHz. Although we are currently targeting H.264 video decoding, the proposed design is well-suited for a number of other DSP applications, such as image filtering, FFT, and audio processing.

3. VIDEO DECODING BACKGROUND
The H.264 codec is currently the most advanced and computationally intensive codec in commercial applications. As such, many older desktop systems do not have the capability to decode H.264 video streams in real-time. This problem is further exacerbated for hand-held (embedded) applications, where the computational performance and power budget is minimal. FreeFood will alleviate the burden of decoding the video from the main processor, exploiting data and memory parallelism.

4. ARCHITECTURAL FEATURES
Video decoding and other multimedia applications are characterized by a high degree of instruction and memory parallelism across loop boundaries. We implemented many architectural features in order to fully take advantage of the characteristics of the targeted code, which will be discussed in the following sub-sections.

To exploit data parallelism, we implemented 4-way SIMD instructions, including 4-way loads and stores to exploit memory parallelism. Data parallelism can be further exploited through additional pipelining, thus improving overall throughput at the expense of latency. To increase the clock speed even further, we decoupled the instruction and data memory units from the main pipeline through use of a stream buffer. We also implement a multiply-accumulator, a popular functional unit for greatly enhancing the performance of multimedia accelerators.

4.1 7-Stage Pipeline
To maximize overall performance at the expense of latency, FreeFood has an aggressive 7-stage pipeline. This pipeline length enables FreeFood to clock at 1.05GHz for a maximum throughput of 4,200 MIPS. However, due to the 3 stages dedicated for instruction execution, dependent instructions must be placed with at least 3 non-dependent instructions in between. We claim that due to the parallel nature of the target code base, this restriction does not significantly affect the overall performance of the microprocessor.

The seven stages perform the following functions:

1. Fetch: Retrieves the instruction from the stream buffer for processing.
2. Decode: Determines the functionality of the instruction.
3. Register File: Reads from the register file.

4. Pre-ALU: Launches the 3-cycle LW, SW, and MAC instructions. Performs ALU pre-computation. Reads/writes to the tri-state bus.
5. ALU1: Performs ALU, shift, and logic operations.
6. ALU2: Finishes the 3-cycle LW, SW, and MAC instructions.
7. WB: Writes the calculated value to the register file.

Section 5 will cover the individual pipeline stages in greater detail.

4.2 Decoupled Memory Banks

Since the slowest function units in FreeFood are the SRAM instruction and data arrays, we decided to decouple the memory units in order to remove the imposed clock speed restriction. The decoupled memory units take 3 cycles to read and write data. The read and write operations are not pipelined.

To compensate for the 3-cycle read, we increased the number of memory banks from one to four, thus enabling up to four concurrent reads and writes. However, the 4-way read and write operations, named vector load and vector store, must modify a single aligned and continuous 4-word block.

To compensate for the 3-cycle memory access, a stream buffer with branch prediction capability was implemented in the front end of the architecture. This performance-enhancing feature will be covered in-depth in Section 6.

4.3 Vector Processing

Vector instructions, or Single Instruction Multiple Data instructions, perform the same operation on different pieces of data. Each vector register contains four 16-bit values. We compute a total of four additions in the VADD instruction: A0 + B0 = C0, …, A3 + B3 = C3. Other instructions follow the same format, performing the same operation on multiple pieces of data. This instruction format enables high code density at the expense of flexibility. However, for our application, flexibility is not a concern due to the repetitive and parallel nature of the code. Higher degrees of parallelism can be extracted by unrolling multiple copies of loop iterations as well.

Besides implementing vector versions of all basic arithmetic operations (add, subtract, shift) and memory operations (load, store), a tristate bus was implemented as a method for communication between the vector and scalar pipelines.

4.4 Multiply-Accumulator

Multimedia and DSP applications often need to perform multiply operations. These resulting values are then usually added together, or accumulated. Thus, to speed up the common case, we implemented a multiply-accumulator (MAC).

The multiply-accumulator is closely modeled to the standard datapath. It features a 4-entry MAC register file, a 4-stage pipelined MAC operation, and register file forwarding. This complexity is necessary to achieve high clock frequencies.

Due to the sheer complexity of the MAC operation, as well as the power and area penalty of synthesized logic, we decided to minimize MAC power consumption by stalling the pipeline when not in use. This simple feature prevents MAC dynamic power consumption when performing other operations, greatly reducing the overall power consumption of the chip.

The MAC operation is available both in scalar and vector format.

5. DATAPATH STAGE DESIGN

In order to support a high clock frequency, FreeFood features an extremely aggressive 7-stage pipeline. The extended pipeline is

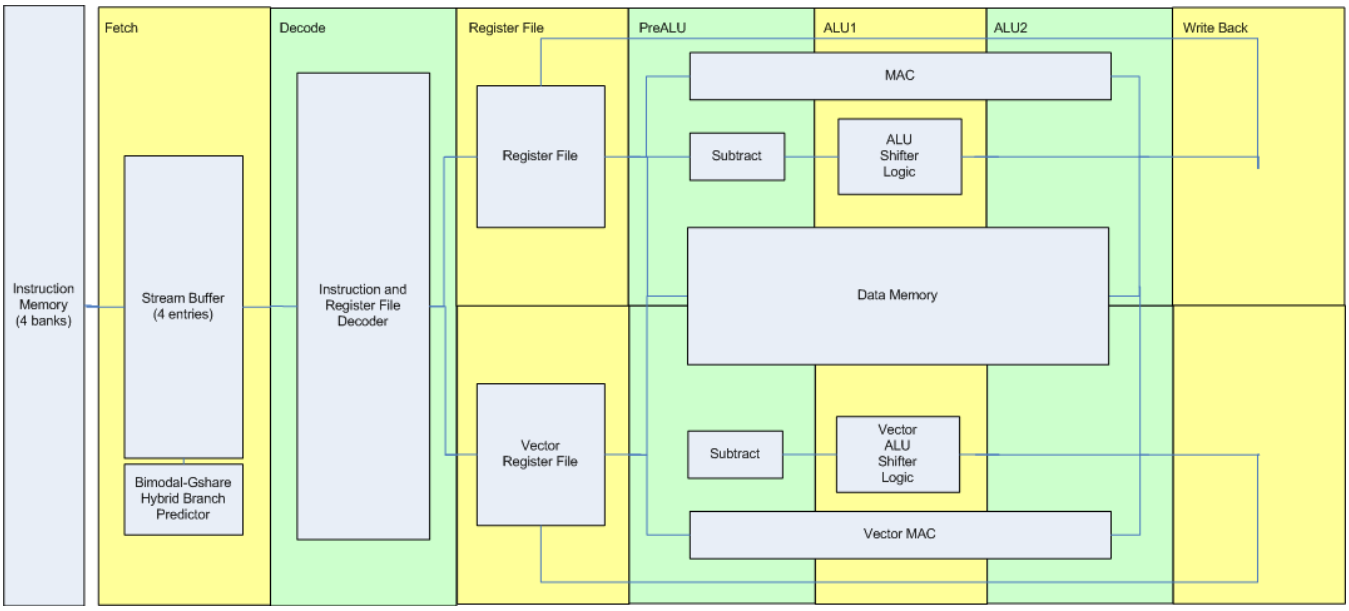

Figure 1. The FreeFood Architectural Pipeline.

based off the standard 4-stage pipeline: Fetch, Decode, ALU/MEM, and Write-Back. Note that in our target ISA, memory operations do not require a base plus immediate value.

To intelligently extend the pipeline, we first predicted the bottlenecks. Two bottlenecks immediately stood out: the SRAM arrays used for instruction and data memory, as well as the synthesized multiply-accumulate logic.

From the specifications, we knew that the SRAM instruction and data arrays will have a delay of up to 1.6ns. Therefore, we allocated 3 stages (3ns) for the load and store operations. The extra overhead is to account for pipeline stage overhead, large wiring capacitances, and arbitration logic.

The MAC operation was also slow, requiring 1.6ns before pipelining. However, due to the MAC needing a special MAC register, we were able to completely decouple the MAC operation from the standard datapath.

Due to the constraints imposed by the memory units and the need to prevent stalling in the pipeline to improve throughput, we decided to extend the ALU operation across three stages. To simplify the forwarding logic and to improve the clock frequency, we neglected to implement forwarding logic for instructions that do not require the full 3 cycles to complete (ALU, logic, shift).

To further improve the clock speed, the register file and the decoder were put into their own separate stages. This was necessary due to the register file needing a 16-bit 1-hot encoded signal for its inputs.

Thus, the 7-stage pipeline implemented in this project consists of the following stages: Fetch, Decode, Register File, Pre-ALU, ALU1, ALU2, and WB.

The final clock speed achieved was 1.05GHz, assuming a clock uncertainty of 100ps. The critical path was within the controller.

5.1 Fetch

The Fetch stage retrieves the instruction from memory for processing. However, since the memory access is split over 3 cycles, a buffering mechanism was necessary to improve overall throughput.

A speculative stream buffer was implemented to improve performance. The buffer predicts the next 4-instruction block with the built-in branch predictor and branch target buffer. With perfect branch prediction, we can successfully hide all the memory latency from the rest of the pipeline.

However, if the branch prediction mechanism misses, then an additional 4 cycle instruction memory miss penalty will be added in addition to the pipeline flush penalty.

5.2 Decode

The Decode stage configures the datapath control signals based on the instruction opcode. The immediate values are sign extended.

Due to the fact that the register file requires 1-hot encoded 16-bit signals, the Decode stage also computes the signals required for the register file. Note that the register file signal decode can be done in parallel with the opcode decode.

5.3 Register File

The register file is the first custom designed pipeline stage of the datapath. It is a 16-word x 16-bit master-slave configuration, with each bit containing one master latch driving one of 16 slave latches. Each latch is controlled by TX gates followed by a bistable element with clock gated feedback to prevent write-in contention. The output of each slave is tied to two read port TX gates for R_{SOURCE} and R_{DEST}, sized for worst case delay of reading from both ports simultaneously. The slave latches are clock gated to minimize power consumption when idle, which comprises of 15/16 slaves for each bit per cycle.

Note that not all instructions use the register file values: some instructions have immediates, and some instructions require 0 or the next program counter (NPC) value. Also, we have register file forwarding, which will select the value currently being written into the register file if the destination register number is equal to at least one of the operands.

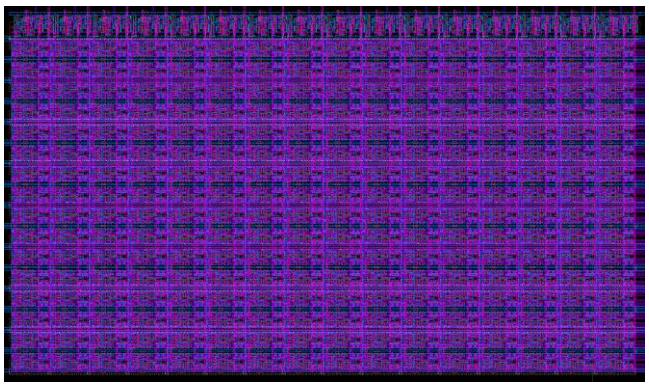

Figure 2. Register File Layout

5.4 Pre-ALU

In the FreeFood architecture, some instructions require three cycles for calculation while others require only one. To prevent pipeline stalls, we have three separate ALU stages. This provides some flexibility for the faster operations.

To simplify the adder during the main ALU stage, we choose between the operand and its negation. This is useful for the subtract opcode, which can be modeled as adding a negative number. However, note that negating a value in 2's complement format involves adding 1 to the inverted value. We can leverage the power of the main ALU to perform this +1 operation. Thus, instead of calculating the complete negation of the operand, we simply calculate the Propagate and Generate values of the first bit and invert all other bits in the operand. These values are passed on through pipeline registers into the ALU1 stage.

For more complex operations, having some form of communication between the vector and scalar pipelines will be necessary. Aside from being able to slowly communicate through the shared data memory, a fast communication path is provided in the form of a tristate bus. This tristate bus links together all the vector and scalar units. This tristate bus drives across 900µm of wire cap, so to account for the drive delay the bus is driven from the pipeline registers at the end of the register-file stage and into the pipeline registers at the end of the Pre-ALU stage.

The three-cycle slow operations for both the vector and scalar datapaths are also launched in the Pre-ALU: LW, SW, and MAC.

5.5 ALU1

During the ALU1 stage, the adder, shifter, and logic units are accessed.

To minimize power consumption and area, we implemented a Sklansky sparse-tree adder. Within the parallel-prefix network of tree adders, the Sklansky adder trades off fanout in exchange for lower power and area. Compared to the Kogge-Stone adder, the Sklansky adder has $1/8^{th}$ of the wiring capacitance but exponential fanout (compared a minimum fanout of 2 in the Kogge-Stone adder). However, the exponential fanout does not greatly affect the output capacitance of the internal nodes. This is due to wiring capacitance dominating the overall capacitance in the design. For example, the final level of the Sklansky adder has a fanout of 8. However, the wiring capacitance on the final level is roughly 30fF. Compare this to the fanout capacitance of roughly 5 fF. With logical effort sizing on the critical path with highest fanout, we reached a delay of ~650ps, well within the 1ns cycle delay tolerance. Compare was also calculated as part of the tree and is needed for branch targets.

The required bitwise logical operations (AND, OR, XOR) were also implemented in this stage using a three nand gate multiplexer to select between the operation. This is a fast and compact design and yielded delays faster than the actual adder.

Finally, a barrel shifter serves as the final custom block in the ALU1 stage. Logical effort sizing and critical path delays were considered to minimize the delay of this block. A fully custom 4:16 decoder was also designed since it was part of the critical path. Since the shifter outputs are driven by PTL NMOS's, these were placed vertically since this results in the shortest wire with least drive capacitance. The critical path delay of the shifter was just over 650ps.

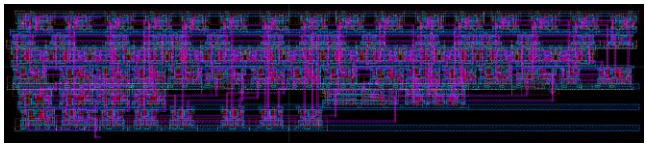

Figure 3. Sklansky Adder / AND / OR / XOR / Compare Layout

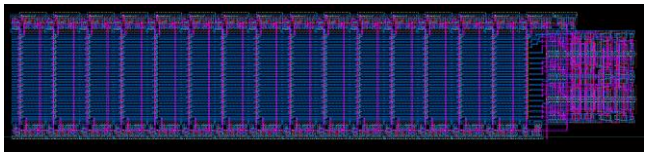

Figure 4. Barrel Shifter with Decoder Layout

5.6 ALU2

The main purpose of the ALU2 stage is to finish the data memory and MAC calculations and to mux between the various calculation results. The value is then stored in the pipeline register in preparation for writing the value back to the register file.

5.7 Write Back

The calculated result is written back into the register file.

5.8 The Pipeline Registers

Due to the low FO4 delay per stage in our design, the pipeline register delay contribution is significant. The pipeline registers were implemented as master-slave latches with TX gate muxes selecting the input sources (up to five MUX inputs). The latch design is identical to the register file latches. The layout was highly compact, with scan-chain imbedded into all the pipeline registers in both the datapaths and controller.

6. DECOUPLED MEMORY BANKS

One of our group's main challenges was finding a method to decouple the slow memory banks without impacting overall performance.

We deemed that an overall throughput of one memory access per cycle was necessary for performance. Therefore, to compensate for the 3-cycle non-pipelined memory access, we duplicated the memory banks 4 times (thus having 4 instruction memories and 4 data memories) as a tradeoff between bandwidth and latency.

Since the SRAM arrays were not pipelined, we could not find a suitable method to allow for more than one LW/SW operation per cycle. While methods exist for dealing with these issues (including duplicating the arrays to allow for multiple reads), we chose to allow for only one LW/SW operation per three cycles instead. This method saves considerable area compared to duplicating the SRAM arrays to allow for multiple outstanding accesses. Plus, since vector loads and stores are able to load or store four values at a time, the overall memory throughput is not impacted compared to a baseline non-decoupled memory pipeline.

A greater challenge was dealing with the front-end instruction memory, since the fetch stage must be able to issue one instruction every cycle. A buffering mechanism is needed to convert the 4 instructions received every 3 cycles into a continuous 1 instruction/cycle instruction stream. A speculative instruction buffer was implemented for this purpose.

6.1 The Stream Buffer

We implemented a stream buffer in the Fetch stage that buffers the instructions coming from the instruction memory. The stream buffer acts as an L0 cache with the added advantage that no tag checks are required.

On reset, the stream buffer is invalid. A request to the instruction memory (I-mem) is sent, which will arrive in 3 cycles. Each I-mem request is for four aligned contiguous instructions. No forwarding was implemented; data from the I-mem must be placed into the stream buffer before use. The stream buffer is validated upon instruction arrival. While FreeFood's pipeline consumes the instructions, the stream buffer speculatively sends the next request to the I-cache. Since the controller can consume a maximum of 4 instructions per cycle, and it takes 3 cycles for the next set of four instructions to arrive, there should theoretically never be any cache misses aside from the first miss.

However, on a branch taken, since the stream buffer does not store any tags, we cannot guarantee that the new branch location is contained within the stream buffer. Therefore, we invalidate the stream buffer and the instruction fetch stage is stalled similar to the state at reset.

To reduce the effect of this penalty occurred with every taken branch, we implemented a branch prediction mechanism with a branch target buffer. Due to the 4-entry instruction block granularity, we can only predict one branch in every aligned 4-entry block. However, multimedia code has less than 1 branch every 4 instructions, so this restriction does not affect performance.

The branch predictor was implemented as a hybrid bimodal-gshare scheme. The bimodal predictor was implemented as a 16-entry non-speculative update branch prediction scheme. The gshare predictor was implemented as a tagged 8-entry non-speculative update with a 3-bit Global Branch History Register. If the tag misses, then the bimodal result is chosen as the branch prediction result. If the tag hits, then the gshare prediction result is used as the branch prediction result.

7. INSTRUCTION EXTENSIONS

With the addition of the vector unit, the controller will need to support many new instructions.

7.1 WAIT

The processor halts until "woken up" by an interrupt signal.

7.2 MACx

Performs a Multiply Accumulate operation, with *X* being a MAC register from 0 to 3. The value of the operation is only written into the MAC register. Available in both vector (VMACx) and scalar (SMACx) form.

7.3 RSTMACx Ry

Resets the MAC register *X*. Writes the value contained in *X* to a designated standard register r0-15. Available in both vector (VRSTMACx) and scalar (SRSTMACx) form.

7.4 VVMOVx Ry, Rz

Utilizes the tristate bus to broadcast a register value *Ry* from vector engine 0-3 (signified by *x*) to all other vector engines. The value is written to vector register *Rz*.

7.5 SVMOV Rx, Ry

Utilizes the tristate bus to broadcast register file value *Rx* from the scalar register file to vector register *Ry*.

7.6 VSMOVx Ry, Rz

Utilizes the tristate bus to transfer from vector register *Ry* with offset *x* to scalar register *Rz*.

7.7 Additional Vector Extensions

Additional vector instructions were implemented as vector versions of the baseline instruction set. The following instructions were added:

- VADD – vector add
- VSUB – vector subtraction
- VAND – vector bitwise AND
- VOR – vector bitwise OR
- VXOR – vector bitwise XOR
- VMOVI – vector move immediate
- VMOV – vector move
- VLSH – vector left shift
- VLUI – vector load upper immediate

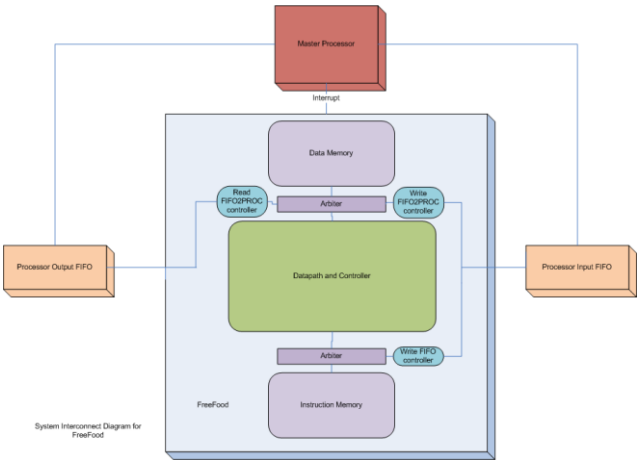

Figure 5. The FreeFood I/O Scheme

In addition, two vector memory operations were added. They function as vector versions of the scalar equivalents, but have the added restriction of memory alignment.

7.7.1 VLOAD (Vector Load)

The VLOAD instruction loads an aligned 4-word (64-byte) block of memory into a vector register. The VLOAD instruction follows the same format as a scalar LOAD instruction.

7.7.2 VSTORE (Vector Store)

The VSTORE instruction stores the contents of a vector register into an aligned 4-word (64-byte) block of memory. The VSTORE instruction follows the same format as a scalar LOAD instruction.

8. SYSTEM IMPLEMENTATION

As shown in Figure 2, FreeFood is designed to be a DSP-type coprocessor that is controlled by a master processor through three 16-bit FIFOs.

We chose the OKI Semiconductor MS81V06160 16-bit high performance FIFO for FreeFood's I/O. This product is designed for providing high-speed, large storage size buffers for use in digital multimedia systems. This FIFO can run at 100MHz, which is the clock speed of the I/O on FreeFood.

The master processor provides FreeFood with the program (placed in the instruction cache) as well as the data (placed in the data cache). When the master processor has finished filling the caches of FreeFood, the master processor sends a reset signal to FreeFood, which then begins execution of the provided program. After the program has finished execution upon reaching the added WAIT instruction, FreeFood goes into sleep mode and sends an interrupt to the master processor. The master processor then processes the data by controlling the read/write FIFOs and modifying FreeFood's instruction and data memories as necessary. Then, the master processor resets FreeFood (note that the reset signal does not reset the values in the register files) and pipeline execution resumes as normal.

The communication between the I/O and memory is handled through the instruction and data arbiters. These synthesized logic blocks arbitrate requests from FIFO, scalar, and vector units. The

arbiters are designed to handle the communication protocols for the OKI Semiconductor device.

9. POWER CONSUMPTION

Component	Power (in mW)	Percentage
Controller	10.64 mW	3.05%
Arbiter	8.5 mW	2.29%
Multiply-Accumulator	44.6 mW	12.79%
Datapath	10 mW estimate	2.87%
512-byte SRAM	10.79 mW	3.09%
Inst/Data Memory	51.66mW	14.81%
Vector Unit	218.4mW	62.7%
Scalar Unit	54.6mW	15.65%
Max Power	290.7mW	83.33%
Max Power with CLK	348.84mW est.	100%

Table 1. Power Consumption Estimates

Since FreeFood is designed as an embedded co-processor, power consumption is a concern. Table 1 depicts power consumption estimates from Synopsys PrimePower, a tool that accurately calculates the power consumption of synthesized logic. We used *high effort* to provide the numbers in this paper. Due to time constraints, we were unable to provide PrimePower with sample stimuli for its power calculations. Therefore, PrimePower relied on random inputs, which grossly overestimates power consumption in certain situations.

For example, the Arbiter logic generally consumes minimal power. However, during FIFO I/O calculations, a much greater amount of the Arbiter activates for communication purposes. PrimePower does not take this into consideration when providing power estimates.

The power consumption numbers for the 512-byte SRAM were provided by the official Artisan datasheets.

To gain a greater understanding of the power distribution in FreeFood, consider the larger modules in Table 1: Inst/Data Memory, Vector Unit, and Scalar Unit. These modules are comprised of multiple basic building blocks.

The Inst/Data Memory consists of four copies of the 512-byte SRAM combined with a single Arbiter.

The Vector Unit consists of four copies of the multiply-accumulator and four copies of the datapath.

The Scalar Unit consists of a single copy of the multiply-accumulator and a single copy of the datapath.

The maximum power consumption number attempts to describe a possible power virus for the FreeFood system. Judging from the power consumption numbers, accessing the memory takes much less power than accessing the multiply-accumulator. Therefore, the VMAC instruction is the most expensive operation, not the memory access. Thus, the maximum power consumption number adds together the controller, the Inst Memory, the Vector Unit, and a single datapath. The single datapath is part of the scalar unit. However, since the datapath does not stall, then its power consumption must be considered in the final value.

Our project does not have a clock tree. Therefore, it is impossible for us to calculate the exact power consumption of this network. Therefore, we estimate power consumption of the clock to be 20% of the total power consumption of the chip. This is reasonable because the final chip is only 1.67mm x 1.58mm and our clock speed is only 1.05GHz (compare this to the 3.7GHz processors on the market today).

9.1 Low Power Logic Families

To minimize power consumption while maintaining high performance, our team used only static CMOS and transmission gate logic families. Dynamic logic was not used due to its high power consumption due to its switching activity and dramatic increase in clock load.

9.2 Sklansky Adder

Research has shown that the Sklansky adder is the best adder choice for EDP. This is due to its minimal logic depth combined with $1/8^{th}$ of the logic and wiring capacitance of the traditional Kogge-Stone tree adder.

9.3 Stalling Multiply-Accumulator

As shown in Table 1, the multiply-accumulate unit uses the greatest amount of power out of all the basic functional units. Therefore, to minimize average power consumption, we stall the multiply-accumulate pipeline registers. This prevents static power from being drawn while the multiply-accumulate unit is not in use.

10. TIMING

The delays for the blocks within the chip are as follows:

Component	Delay	Units
Controller	850	ps
Arbiter (MUX)	340	ps
Arbiter (FIFO)	700	ps
MAC	710	ps
Adder	646	ps
Shifter	659	ps
Register File (T_{CQ})	407	ps
Pipeline T_{SETUP}	150	ps
Pipeline T_{CQ}	158	ps

Table 2. Component Delay Estimates

Based on the above timings along with 100ps tolerances for clock uncertainty, FreeFood is estimated to run at 1.05GHz, with the bottleneck being in the Controller. The datapath critical path was in the ALU1 stage:

$$T_{CQ}\ (158ps) + T_{ADDER}\ (646ps) + T_{SETUP}\ (150ps) + \text{clock uncertainty}\ (100ps) = 1054ps.$$

However, the following stage, ALU2, consists of no logic outside of the MAC, which executes in parallel with the datapath. Therefore, we borrowed 200ps from this stage to remove the critical path. Unfortunately, we cannot do the same for the synthesized controller. Thus, the ultimate critical path is in the

Controller. A custom controller may allow us to achieve even higher frequencies pushing 1.25GHz.

11. GLOBAL LAYOUT

The layout of the entire chip is depicted in Figure 3. The chip is LVS and DRC clean. The area is 1.67mm x 1.58mm, designed on the IBM 130nm standard Vt 1.2v process. Since all four vector engines share the same inputs, we can reduce wiring by stacking the vector units side by side. The scalar units need slightly different inputs. Thus, we place the scalar unit on the other side of the controller. We were able to minimize the wiring by placing the burden on the arbiter: we placed the inputs and outputs of the arbiter such that we could connect each datapath to the arbiter with straight wires. Note that each vector engine consists of a scalar datapath plus a scalar multiply-accumulator.

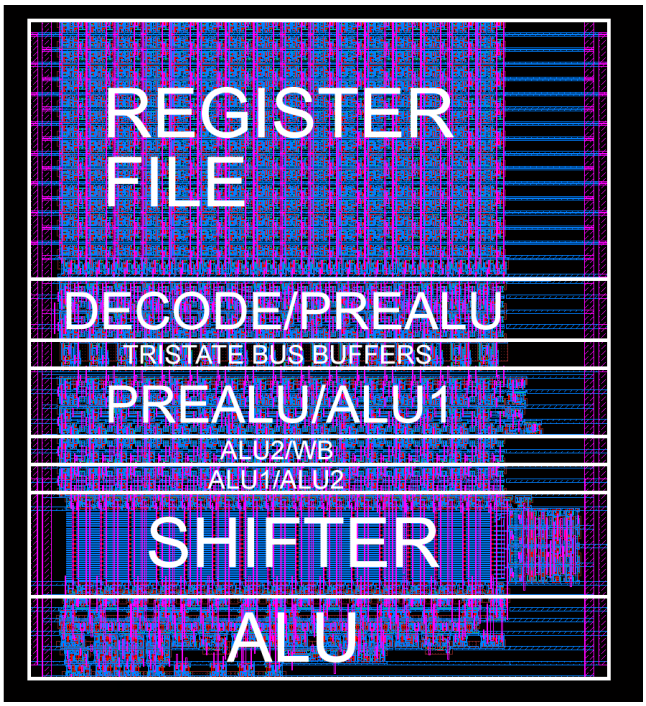

Figure 6. The Scalar Datapath Layout

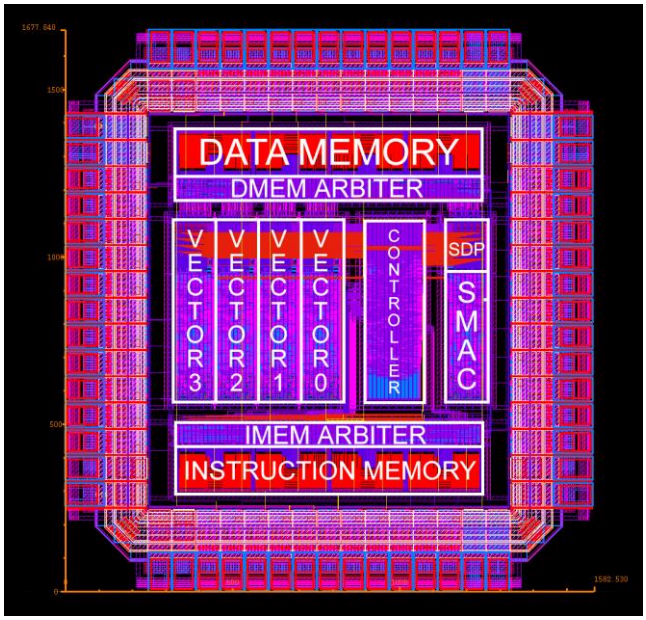

Figure 7. The Layout of FreeFood

(SDP = Scalar Datapath, SMAC = Scalar MAC)

12. CONCLUSION

We designed FreeFood, a 1.05GHz DSP with vector extensions capable of 4200 MIPS. FreeFood is power efficient, consuming only 348mW in the worst case. Many other complex and novel pipelining schemes are implemented in order to hit our targeted speed. FreeFood's main target is to provide uncompromising performance while maintaining reasonable power efficiency.

13. ACKNOWLEDGMENTS

Our thanks go to Joel VanLaven and Wei Hsiang Ma for providing extremely useful advice for all areas of our project.

14. REFERENCES

[1] Peleg, A.; Weiser, U., "MMX technology extension to the Intel architecture," *Micro, IEEE* , vol.16, no.4, pp.42-50, Aug 1996

[2] Wiegand, T.; Sullivan, G.J.; Bjntegaard, G.; Luthra, A., "Overview of the H.264/AVC video coding standard," Circuits and Systems for Video Technology, IEEE Transactions on vol.13, no.7, pp. 560-576, July 2003

[3] "Draft ITU-T recommendation and final draft international standard of joint video specification (ITU-T Rec. H.264/ISO/IEC/AVC)," Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, JVT-G050, 2003.

[4] Jih-Ching Chiu; I-Huan Huang; Chung-Ping Chung, "Design of instruction stream buffer with trace support for X86 processors," Computer Design, 2000. Proceedings. 2000 International Conference on , vol., no., pp.294-299, 2000

[5] Patil, Dinesh; Azizi, Omid; Horowitz, Mark; Ho, Ron; Ananthraman, Rajesh, "Robust Energy-Efficient Adder Topologies," Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on , vol., no., pp.16-28, 25-27 June 2007