A STREAMLINED DSP MICROPROCESSOR ARCHITECTURE

Michael Takefman* and Paul Chow
Dept. of Electrical Engineering, University of Toronto
Toronto, Canada

ABSTRACT

This paper describes a microprocessor architecture for Digital Signal Processing. The architecture offers almost twice the performance of the Motorola DSP56000 microprocessor, while maintaining assembly-code compatibility. Improved performance is achieved by streamlining the instruction set, using a seven-stage pipeline, and adding a second memory write stage late in the pipeline.

1 INTRODUCTION

The methodology for Reduced Instruction Set Computer (RISC) design has significantly increased the performance of General Purpose (GP) microprocessors [1, 2]. This paper describes the first results of the application of RISC principles to the design of a microprocessor architecture for Digital Signal Processing (DSP) [3]. This research investigated an existing DSP microprocessor, profiling the execution of programs, and examining the effects of deeper pipelines on the performance of the instruction set. The result is a streamlined version of the Motorola DSP56000 microprocessor using a seven-stage pipeline that can execute an instruction each clock cycle, doubling the performance on a per clock basis.

2 METHODOLOGY

We studied an existing microprocessor architecture for two reasons. The primary reason was the availability of application source code. A tenet of RISC design is to make design decisions based upon the impact of design alternatives on the performance of real programs. We did not have access to a retargetable optimising compiler, so we could not begin with applications written in a high-level language, and we did not wish to hand-code all of the benchmarks. By selecting an existing microprocessor, we could use available libraries of hand-optimised code. The second reason for studying an existing microprocessor was to determine the viability of our ideas for improving performance before exploring architectures that would also be more efficient targets for compilers.

The Motorola DSP56000 microprocessor was chosen as the base architecture for the following reasons: it was a recent architecture; it contained all the features required for use as an efficient DSP processor; it had a high degree of internal parallelism[4, 5]. Furthermore, source code was readily available from Motorola's Dr. Bub bulletin board service, and Bell Northern Research made some of its application code available for analysis[1].

A fully-metered high-level simulator of the DSP56000 was written in C++[6]. Fourteen programs were selected as benchmarks and were run using the simulator to profile execution. The benchmarks include: FFTs, FIR filters, IIR filters, a Lattice filter, an LMS adaptive FIR filter, matrix multiplication, Reed Solomon encoding, RSA encryption and a Sliding Window Correlator. The profiling results were used to design an initial experimental architecture. The experimental architecture was captured as a high-level simulator written in C++. To verify the correctness of the simulation results, the states of the simulators were checked against the DSP56000 simulator provided by Motorola. The benchmark programs were then run on the experimental simulator. Performance bottlenecks were identified and the simulator was modified as the architecture evolved over several iterations. The final iteration was designated X56k and a floorplan of the chip was proposed.

3 DSP56000 REVIEW

To understand the differences between the DSP56000 and X56k, a brief review of some of the DSP56000 features is presented below. The interested reader will find full details in the DSP56000 manual [7, 8]. An important feature of the DSP56000 is the parallelism and memory bandwidth available internally. Each instruction cycle (two clock cycles) the processor can perform up to three memory accesses (one program, two data), a multiply/accumulate operation using the Data ALU, and two addressing operations using the Address Generation Unit (AGU). The microprocessor is organised as a three-stage pipeline (instruction fetch, decode and execute), with each stage operating in a minimum of two clock cycles. Some instructions require greater than two cycles at certain pipeline stages.

The DSP56000 is not a pure load/store machine. All Data ALU and AGU instructions only operate on register operands, but some of the other instructions can specify memory operands as well. We refer to instructions that utilise memory operands as miscellaneous instructions. The miscellaneous instructions include branch, zero-overhead looping, and bit-manipulation instructions. Bit-manipulation instructions can also cause indivisible read/modify/write memory cycles. As well, some of the memory-move instructions perform a read from one memory location and a write to another location in one instruction.

*Michael Takefman is now with Bell Northern Research, Ottawa, Ont.

[1]Our thanks to Messr. B. O'Higgins, M. Wiener, A. Wierich and Q. Meek for their help.

CH2977-7/91/0000-1257 $1.00 © 1991 IEEE

An important feature of the DSP56000 pipeline is that there are no load/store delays when accessing Data ALU registers, and there is a single instruction cycle load delay when writing registers in the AGU. Thus, any value created in the Data ALU in the current instruction can be written to memory in the next instruction, and any value read from memory to a Data ALU register can be used in the next instruction.

The Data ALU contains two 56-bit accumulators and four 24-bit input registers. The AGU contains two register banks, each containing 12 16-bit registers organised as four register triples (Rn, Nn, Mn). The R register is used to reference a memory operand. The N register provides an offset to be added to R in address calculations. The M register value selects the type of arithmetic used in the address calculation (linear, modulo, or FFT). Finally, there are a number of registers used for hardware looping and other microprocessor control functions.

4 X56K ARCHITECTURE

A block diagram of the X56k architecture is shown in Figure 1. The hashed lines indicate bypass paths between the functional units. It is a load/store architecture with four logical pipelines. The AGU pipeline performs memory operand address calculations while the Memory pipeline performs the accesses. The Data ALU pipeline performs multiply/accumulate and other ALU operations, and the Program Control Pipeline (PCP) performs all other instructions (branches, loops, bit tests etc.). Bypass logic is used extensively within each pipeline to remove pipeline dependencies. Bypass logic is also used to bypass values to and from the Memory pipeline, and the other pipelines. The four logical pipelines are partitioned as three functional units, namely the AGU, the Data ALU and the PCP. The Memory pipeline is distributed and portions of it are replicated across the three functional units. The replication of the memory pipeline causes an increase in the area of each pipeline, but decreases the amount of global routing (for bypassing) between functional units. The first two stages of the pipeline are shared among the logical pipelines and the longest pipeline is seven stages.

4.1 Instruction Streamlining

Statistics were gathered to determine how the DSP56000 instruction set was being utilised. The measurements revealed that the use of memory operands by the miscellaneous instructions were infrequent enough to justify their removal, making X56k a true load/store machine. Furthermore, instructions that required two memory operations (peripheral memory move, and read/modify/write) were also rarely used. Table 1 shows the instruction distribution across the benchmarks. For each benchmark the following information is given: the number of instructions executed, the number of Data ALU instructions as a percentage of all instructions and the percentage of Data ALU instructions with a Move operation in parallel. The number of NOPs (generally parallel Move operations with no available Data ALU operation), non-parallel Move instructions, branch instructions and other miscellaneous instructions are given as a percentage. This table shows that the branch and miscellaneous instructions are executed infrequently. Table 2 shows the operand usage for the instructions. The Cnt column denotes the number of instructions that can use memory operands. The Reg column lists the percentage of instructions that use a register operand. The Mem column denotes the use of a memory operand, and Lit lists the usage of literal operands. The table indicates that the majority of the operands are held in registers or within literals. Therefore, designing X56k as a true load/store machine and removing instructions that require multiple memory operations would not have a disastrous effect on performance. DSP56000 instructions that are not implemented directly in the X56k instruction set are implemented using multiple X56k instructions. The expansion is handled automatically by the X56k simulator during execution[2]. There is no reorganisation performed on the resulting code. Therefore, our performance estimates represent a lower bound on performance.

4.2 Memory Pipeline

The most unique feature of the X56k architecture is the memory pipeline. The DSP56000 has a high peak memory bandwidth, and a number of the benchmarks utilise almost all the available bandwidth. X56k uses a deep pipeline to achieve single-cycle operation with a reasonable clock period. Unfortunately, due to the small number of Data ALU accumulator registers, results computed in the ALU are often written to memory within the next two instructions to free the registers for reuse. With a deep pipeline, dependencies prevent the ALU result from being used in either of the next two instructions. The obvious solution is to add more registers. A larger register set would decrease the frequency of register reuse, and would make it possible to schedule the code so that a store instruction could be delayed until the result is available. However, we did not add any additional Data ALU registers to X56k because of our constraint of being code compatible.

The solution used is the addition of a second memory write-back stage late in the pipeline. The MEM_2 stage is located in parallel with the accumulator (ALU) stage of the Data ALU pipeline as shown in Figure 1. Consider the case where an instruction creates a result (in the accumulator) and one of the next two instructions writes the result to memory. Without the MEM_2 stage NOPs must be inserted to resolve the pipeline dependency. The MEM_2 stage allows the program to execute without the addition of NOPs because the ALU stage forwards the result to the MEM_2 stage. If the memory write operation occurs after two instructions, the bypass logic forwards the result to the MEM_1 stage. Unfortunately, a resource conflict occurs if the MEM_1 stage and the MEM_2 stage both attempt to access the same memory bank. In this case, the operation is serialised and the operation of the MEM_1 stage and the microprocessor are stalled while the MEM_2 stage completes its operation first. Bypass logic is used within the memory pipeline to handle dependencies, and between the memory pipeline and the other pipelines. The bypass logic must also handle the case where an instruction is scheduled to write a value in MEM_2 when the next instruction is reading the same memory location in MEM_1. In this case, the result is forwarded by the ALU stage to the MEM_1 stage and then to the MEM_2 stage.

[2] An X56k assembler would take DSP56000 assembly code and treat the unimplemented instructions as macros.

4.3 PCP Pipeline

The PCP is five stages long and includes the instruction fetch and decode stages. The PCP is responsible for executing all instructions that do not use the AGU or Data ALU pipelines such as branches, loops, bit tests and microprocessor control instructions. The MISC stage actually performs the operations. The DSP56000 uses the Global Data Bus to transfer data between functional units to avoid conflict with the internal memory busses. X56k uses the same bus to transfer data to the PCP from the register files of the other pipeline units. The result is returned to the source pipeline using one of the memory buses during the MEM_1 (Delay) stage.

4.4 AGU Pipeline

The AGU pipeline is five stages long including the instruction fetch and decode stages. The register read operation occurs simultaneously with decoding. The AGU uses two stages to perform the addressing arithmetic. The AGU_1 stage performs linear and FFT addressing. AGU_1 also performs modulo addressing if the offset is one. In the case where the offset is not one, the AGU_2 stage performs the additional add operation to complete the modulo operation. If the AGU_2 stage is required for an address calculation, the next instruction cannot reference the result register of the instruction in the AGU_2 stage. Bypass logic removes all other pipeline dependencies. If the offset addressing mode is used in conjunction with modulo arithmetic (which was not used in our benchmarks), multiple instructions are required to correctly access the memory operand. To reduce the area required for bypass logic, only the R registers are bypassed from the memory pipeline. Profiling results showed that it was rarely necessary to bypass N or M register values to the AGU pipeline. To reduce area in the register file, only a single writeback port is provided. This restricts the use of move operations that utilise an addressing mode that updates an address register. The destination register of such a move instruction cannot reside in the same register file. Profiling results showed that this instruction was never executed. Both of these results indicate that the area saved causes very little performance degradation.

4.5 Data ALU Pipeline

The Data ALU pipeline is seven stages long including the instruction fetch and decode stages. Two stages are allocated for the multiplier and one stage is allocated for the accumulate/ALU operation. The latency of the multiplier/accumulator causes pipeline dependencies when an ALU result is required as a multiplier input in a subsequent instruction. Profiling results show that for a three-stage multiplier/accumulator no such dependencies occur in our benchmarks. However, pipeline dependencies would occur for a four-stage multiplier/accumulator. Normally, the assembler would check for such dependencies and insert the appropriate NOPs (if required).

4.6 Inter-Pipeline Dependencies

One performance bottleneck that cannot be removed without extensive global routing occurs when the results from one pipeline are used as the source operands in another pipeline (excluding the memory pipeline). The most glaring example of this is the Reed-Solomon Encoder (rs_enc), whose performance is only a factor of 1.36 better than on the DSP56000. The rs_enc program is essentially a table look-up routine where the Data ALU is used to create an index into a table, and the AGU is used to perform the addressing operation. Due to the deeper pipeline used in X56k, NOPs must be inserted in the code to resolve the inter-pipeline dependencies. The Sliding Window Correlator suffers minor performance loss due to the same type of dependencies. The RSA Encryption program suffers some performance loss due to dependencies between the AGU and the PCP. It is possible that code reorganisation could remove some of the NOPs, reclaiming some of the lost performance. This was not investigated.

5 PERFORMANCE

Table 3 shows the performance of the X56k architecture compared to the DSP56000. The first two columns give the number of clock cycles required to complete the program execution. The first column is labelled D56k and gives the results for the DSP 56000, and the second column gives the results for X56k. The next column gives the ratio of X56k cycles to DSP56000 cycles. The next four columns indicate the number of cycles lost to code expansion (Extra), inter-pipeline conflicts (Conflicts), memory resource conflicts between MEM_1 and MEM_2 (Mem), and addressing mode dependencies in the AGU (AGU).

The FIR filters achieve speedups greater than two due to a more efficient implementation of the zero-overhead loop instructions. The FFTs lose the majority of their performance due to memory conflicts. However, without the use of the second memory write stage the speedup falls below unity. In fact, with the exception of the rs_enc and rsa_192 benchmarks, memory conflicts account for the majority of lost performance.

6 CONCLUSIONS

The X56k architecture can deliver over twice the performance of the DSP56000 on our set of benchmarks while maintaining assembly code compatibility. The improvement is achieved through instruction streamlining, and the use of a seven-stage pipeline, single-cycle operation, and a second memory writeback stage. It is estimated that this can be done with about 30% more hardware and no effect on the cycle time. Further development of X56k should investigate a much wider set of real application programs.

This research has shown that deep pipelines are a viable method of improving the performance of a DSP architecture, and that there are many opportunities for even better performance if there is no constraint on code compatibility to an existing architecture. Possible directions to explore would be the addition of more registers, simpler instruction sets, and the use of advanced optimising compiler technology to help design better instruction sets.

REFERENCES

[1] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 1990.

[2] J. Hennessy. VLSI Processor Architecture. *IEEE Transactions on Computers*, C-33(12), December 1984.

[3] M. Takefman. Improving the Performance of a DSP Microprocessor Architecture. Master's thesis, University of Toronto, 1990.

[4] E. Lee. Programmable DSP Architectures: Part I. *IEEE ASSP Magazine*, 5(4):4–19, October 1988.

[5] E. Lee. Programmable DSP Architectures: Part II. *IEEE ASSP Magazine*, 6(1):4–14, January 1989.

[6] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[7] K. Kloker. The Architecture and Applications of the Motorola DSP56000 Digital Signal Processor Family. In *IEEE International Conference on Accoustics Speech and Signal Processing*, pages 523–526, April 1987.

[8] Motorola. *DSP56000/DSP56001 Digital Signal Processor User's Manual*, 1988.

We would like to acknowledge the support received from the Motorola DSP University Support Program, Bell Northern Research, URIF and NSERC grant OGP0036648. Michael Takefman was supported by an NSERC Postgraduate Scholarship and a Bell Northern Research Postgraduate Scholarship.

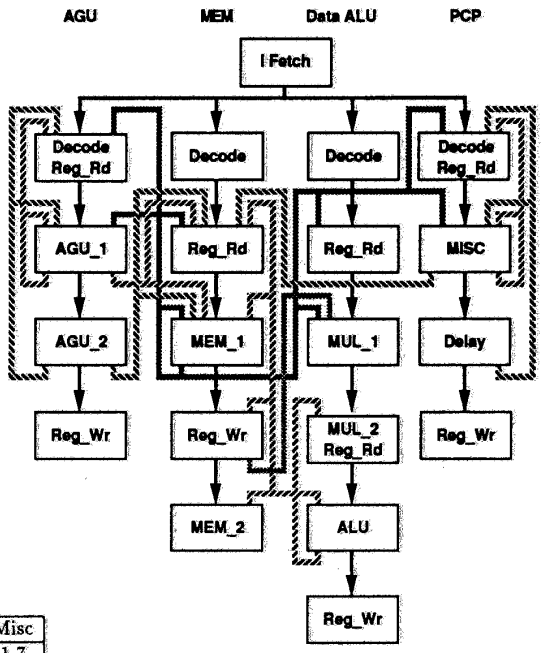

Figure 1. X56k Pipeline Block Diagram

Benchmark	Instrs	ALU	// Move	NOP	Move	BRA	Misc
fft_1024	32354	93.0	96.9	1.7	3.6		1.7
fft_256	8067	76.44	82.1	3.5	16.9		3.3
fir_256_1	268	95.9	99.2	1.1	2.2		0.8
fir_32_64	2312	91.4	99.9	0.1	5.7		2.8
iir_2	3769	61.8	54.0	1.0	35.2	0.1	2.0
iir_3	265	63.4	68.3	5.3	29.4	0.8	1.1
iir_4	406	53.2	73.7	15.0	19.0	0.5	12.3
latnrm_64	1736	62.7	77.2	11.2	18.7		7.4
lmsfir_256	1042	49.3	49.6	0.5	50.0		0.2
mat_13_33	19	47.4	56.3	10.5	42.1		
mat_33_33	102	35.3	36.5	5.9	46.1		12.8
rs_enc	4217	34.7	42.0	16.8	47.9		0.5
rsa_192	113162	69.3	80.8	4.6	16.1	2.8	7.2
swc_32_256	9753	91.9	94.4	0.1	5.3		2.6

Table 1. Instruction Distribution of Benchmark Programs

Benchmark	Cnt	Reg	Mem	Lit
fft_1024	547	98.7		1.3
fft_256	266	98.9		1.1
fir_256_1	2			100.0
fir_32_64	65			100.0
iir_2	26	92.3		7.7
iir_3	2			100.0
iir_4	2			100.0
latnrm_64	129			100.0
lmsfir_256	2			100.0
mat_13_33	0			
mat_33_33	13			100.0
rs_enc	22			100.0
rsa_192	10079	71.1	23.2	5.6
swc_32_256	257			100.0

Table 2: Operand Usage of Miscellaneous Instructions

Benchmark	M56k	W56k	Speedup	Extra	Conflicts	Mem	AGU
fft_1024	66248	44311	1.50	0	336	9212	2341
fft_256	16694	10430	1.60	0	64	2295	0
fir_256_1	544	271	2.01	0	0	0	0
fir_32_64	5010	2378	2.11	0	0	63	0
iir_2	7588	4157	1.83	0	0	360	23
iir_3	532	317	1.68	0	0	47	0
iir_4	814	435	1.87	0	0	24	0
latnrm_64	3730	1866	2.00	0	0	127	0
lmsfir_256	2090	1046	2.00	0	0	0	0
mat_13_33	38	25	1.52	0	0	2	1
mat_33_33	230	114	2.02	0	0	0	9
rs_enc	9724	7149	1.36	0	2496	408	24
rsa_192	243416	127436	1.91	1395	4996	2289	517
swc_32_256	20022	11300	1.77	256	1028	0	0

Table 3: X56k Benchmark Performance