AN ABSTRACT OF THE THESIS OF

Jumnit Hong for the degree of Master of Science in Electrical and Computer Engineering presented on June 17, 2004.
Title: Development and Evaluation of X32V

Abstract approved:
Redacted for privacy

Ben Lee

Embedded processors are utilized in many applications with considerable time spent developing and maintaining functionality and performance. Performance being a key factor in adding features such as video and audio to a product. Configurable processors, such as X32V, allow the addition of functionality and performance without large increases in design time. This thesis presents X32V as a viable embedded processor architecture capable of competing with current processor architectures.

X32V is an embedded processor architecture developed at Oregon State University. It features a highly dense instruction coding format, integrated conditional branching, and configurability. With performance similar to an ARM SA-110 CPU, X32V provides a promising future to developers seeking a high performance and configurable embedded processor.

Development and Evaluation of X32V

by
Jumnit Hong

A THESIS

submitted to

Oregon State University

in partial fulfillment of the
requirement for the
degree of

Masters of Science

Presented June17, 2004
Commencement June 2005

Master of Science thesis of Jumnit Hong presented on June 17, 2004.

APPROVED:

Redacted for privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for privacy

Director of the School of Electrical Engineering and Computer Science

Redacted for privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

Jumnit Hong, Author

ACKNOWLEDGEMENTS

The author expresses sincere appreciation for the following individuals. Dr. Ben Lee for his help in school, research, and life. David Zier for his work on EM3 and X32V. Jarrod Nelson for his expert assembly programming skills. Savrithi Venkatachalapathy for her determination in creating a compiler for X32V.

Table of Contents

List of Figures

List of Tables

Development and Evaluation of X32V

1 Introduction and Background

1.1 Introduction

There are many different classes of embedded processors currently available. Some are single cycle machines, executing one instruction in one cycle, some are pipelined, allowing faster cycle times and multi-cycle instructions, and finally there are very high performance superscalar, out-of-order execution embedded processor cores. Choosing an architecture that provides the best value can be complicated and is critical in creating a product that is profitable. Companies spend large amounts of money developing products and would like to spend as little as possible to maintain and upgrade the product. Therefore System-on-a-Chip (SoC) designers creating products with short lifecycles, ideally would be able to add functionality and performance using the same base architecture it was developed under.

For most embedded systems a pipelined architecture provides the best price to performance ratio since pipelined processors provide high performance without large increases in die size. Adding functionality like floating point and single instruction multiple data (SIMD) features to a pipelined processor can provide increased performance and a longer product lifecycle without major increases in costs. Embedded systems are typically used in portable applications requiring nominal power consumption. Typically a pipelined processor consumes less power than superscalar processors, providing a good choice for portable applications. Therefore, a pipelined processor provides the best architecture for creating an embedded processor.

This thesis discusses a configurable embedded processor called X32V and evaluates its performance with respect to the Intel StrongARM processor that implements the SA-110 core.

1.2 Background

To understand how performance is evaluated between the ARM and X32V instruction sets an overview of 5-stage pipeline processors is needed since SA-110 and X32V are both 5-

stage RISC pipelines. Also compilers will be discussed along with how performance is determined without actual hardware.

1.2.1 Processor Architecture: 5-Stage Pipelined Processors

A pipelined processor executes instructions in an overlapped manner with portions of instructions executing at one time. For example, take the classic analogy of doing laundry, one stage is the washer, the second the dryer, and third is folding. One way to do the laundry is to have a load complete all three stages before starting a new load. Using this method, a load of laundry will have to be washed, dried, and folded before another load will be able to start. This is highly inefficient, the resources available are not being fully utilized. A load of laundry can be in the washer while another is being dried in the dryer and even another load can be folded all at the same time. This allows three loads of laundry to be in the "pipeline" at any one time. Rather than wait for one load to complete all three stages before starting a new load, three loads can be partially performed at a time. Ideally the speedup from pipelining equals the number of pipe stages.

Instructions in a pipeline are executed in-order and should always stay in order until the last stage. This gets complicated when multi-cycle instructions are introduced. Additional hardware is needed to keep track of in-order execution and is usually implemented through a reservation shift register (RSR).

In a typical 5-stage pipeline the stages are instruction fetch (IF), instruction decode (ID), execute (EX), memory accesses (MEM) and write-back to register file (WB). These stages are described below:

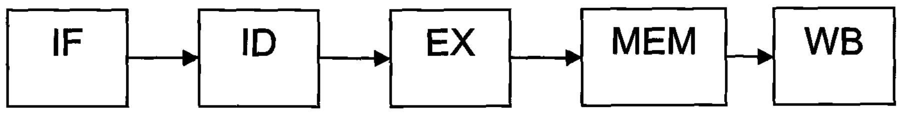

Figure 1: A typical 5-stage pipeline

Instruction Fetch (IF)

In a Harvard architected machine, memory is split into two portions one being for instructions and one being for data. During the instruction fetch stage an instruction is brought into the pipeline from the address that the current program counter holds.

Instruction Decode (ID)

During the ID stage the instruction is decoded to provide the type of operation going to be performed and to set the ALU accordingly. Also during this stage the operands are determined from the decoded instruction and indexed into the register file.

Execute (EX)

In the EX stage the instruction is actually run through the ALU and computed.

Memory Accesses (MEM)

During the MEM stage accesses to memory are performed through loads and stores. This is where the processor interacts with data memory bringing in values and writing out values.

Write-back (WB)

During the WB stage the register file is updated with the result of the instruction.

1.2.2 Compiler, Assembler, Linker

Performance is not only determined by the architecture of the processor itself, the compiler plays a crucial role in creating the most efficient instructions for the processor to execute. Compilers typically take a high level language, such as C, and create assembly code. Then the assembler is used to make a binary file from the assembly code. The binary files are linked together to form an executable binary file that executes on your host system.

Compilers implement many advanced techniques to extract Instruction Level Parallelism (ILP) to create efficient code. ILP is the overlap of executing instructions in a pipelined processor. [2] explains many of the techniques used in compilers to exploit ILP.

1.2.3 Simulators

Creating actual hardware to test new architectural improvements is a time consuming and expensive venture. Extracting statistics such as cycle counts and hit-rates are very difficult to perform in actual hardware. So, to test if concepts operate according to conjecture, processor simulators are often used to simulate actual hardware. Simulators are typically written in a high level language such as C or C++. In simulators anything can be assumed, such as infinite system memory, a perfect branch predictor, a divide unit that performs all operations in one cycle, and so on. Having this flexibility allows creativity in designing new architectural improvements.

There are two different types of processor simulators, functional simulation and cycle-accurate simulation. Functional simulators execute one instruction at a time and are mainly used to validate the functionality of an ISA. The exact hardware is not modeled accurately and there are no timing statistics. In a cycle-accurate simulator the actual hardware is modeled as accurate as possible to achieve precise statistics. Therefore a cycle-accurate simulator is much more complex than a simple functional simulator. Table 1 shows several simulators available from the SimpleScalar toolset and their complexity and performance levels, information was obtained from [3].

Compilers play a crucial role in creating a comprehensive simulation tool. They provide the compilation tools necessary to create benchmarks programs that run on simulators.

Simulator	Description	Lines of code	Simulation Speed
sim-safe	Simple functional simulator	320	6 MIPS
sim-fast	Speed optimized functional simulator	780	7 MIPS
sim-outorder	Detailed Micro-architectural timing model	3900	0.3 MIPS

Table 1: SimpleScalar baseline simulation models

1.3 SimpleScalar

SimpleScalar is a widely used out-of-order processor simulation toolset developed by Todd Austin while conducting his doctorial thesis at the University of Madison-Wisconsin. The ISA simulated in SimpleScalar is called the Portable Instruction Set Architecture (PISA). This ISA is constructed of 64-bit instructions similar to the MIPS instruction set. There are three instruction formats: register, immediate, and jump. Figure 2 diagrams the PISA instruction formats.

There are several simulators available in SimpleScalar including: sim-safe, sim-fast, and sim-outorder. sim-safe and sim-fast are functional simulators while sim-outorder is a cycle-accurate simulator. The simulator core is complemented by many necessary modules such as cache, memory, statistics, loader, and system call modules. The cache module instantiates multiple level cache's including L1 and L2. Each cache level can be assigned a certain configuration, hit latency, and size. The memory module holds the program binary that is loaded by the loader module. Statistics are gathered by the statistics module. Information such as total number of instruction executed, number of cycles, hits and misses on cache's, etc. are gathered and printed to the screen when the simulation has completed. Since there is no operating system running on the simulated processor a system call handler is needed for programs to write and read to files. The system call handler intercepts system calls requested by the executing program and makes the corresponding call to the host systems operating system. Then returns the results of the system call to the executing program.

Register Format

16	16	8	8	8	8
Annote	Opcode	rs	rt	rd	ru/shamt

Immediate Format

16	16	8	8	16
Annote	Opcode	rs	rt	imm

Jump Format

16	16	6	26
Annote	Opcode	Unused	target

Figure 2: SimpleScalar instruction formats

Applications executed on SimpleScalar run on an execution-driven simulation core. An execution-driven simulation core allows precise modeling of actual computer hardware. The instruction set is defined separately from the hardware simulation core to make it portable to other ISA's. Each instruction is interpreted by the instruction set emulator which in turn directs the operation of the hardware simulator.

To make SimpleScalar as flexible as possible configuration files are used to describe the internal pipeline, cache, memory, and branch options. Cache options such as size, latency, and replacement policy are determined in the configuration file. The pipeline has the option to perform each instruction in-order which later will be important to simulate an in-order pipelined processor. Using different configuration files will result in different performance statistic gathered by SimpleScalar, allowing SimpleScalar to be tuned to perform similarly to an actual hardware processor.

SimpleScalar has been retargeted for several ISA's which include Alpha, PowerPC, x86 and ARM. ARM is most important in this thesis so it will be discussed in further detail in the next chapter.

2 ARM

2.1 Introduction

ARM is a popular embedded processor architecture used widely around the world and implemented in many different applications. The acronym stands for Advanced RISC Machine. It features a load-store architecture, a fixed-length 32-bit instruction set, and 3-address instruction formats. The main concern of the ARM designers was to make ARM as simple as possible. An in depth look at ARM is taken because X32V is compared extensively with ARM later in this thesis.

2.2 ARM Architecture and ISA

The ARM architecture is similar to many embedded processors, such as having 16 registers, simple RISC instructions, and exception handling. But there are many unique features that ARM employs to create higher code density and better performance. The unique features of the ARM architecture are listed below:

- a load-store architecture
- 3-address instructions
- conditional execution of every instruction
- load and store multiple registers
- ability to perform a shift and ALU operation in a single instruction that executes in a single clock cycle
- Thumb instruction set that compresses instructions into 16-bits

Since ARM implements a load-store architecture all instructions will process data already in its register file. Memory accesses are completed only through load and store instructions that read or write values into and out of the register file. Therefore all ARM instruction fall into three categories: data processing instructions, data transfer instructions, and control flow instructions.

3-address instructions force the two source operand registers and result register to be explicitly defined in the instruction format.

All ARM instructions can be conditionally executed depending upon the N, Z, C, and V flags located in the CPSR. N is negative, Z is zero, C is carry, and V is overflow. These condition bits are the results of the last ALU operation and stored in the Current Program Status Register (CPSR). As seen in Figure 3 the most significant 4-bits of each instruction determines how the instruction will be conditionally executed. Table 2 shows each conditional option available to each instruction. Using this method of conditional execution of each instruction allows simple implementation of if-then-else statements.

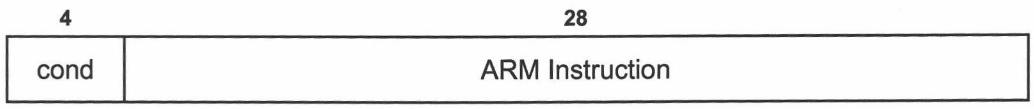

Figure 3: The ARM condition code field

COND [31:28]	Mnemonic extension	Interpretation	Status flag and state of execution
0000	EQ	Equal/equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never	none

Table 2: ARM condition codes

Loading and storing multiple registers allows any of the 16 registers to be loaded or stored to memory. These type of instructions are mainly used to store information when entering a function to save workspace data. They are also useful for high-bandwidth memory block copies to memory.

Each data processing instruction has the ability to perform a shift and ALU instruction in one cycle. The shift instruction is performed before it enters the ALU. A common example utilizing this feature is a shift and add instruction. A data processing instruction example is seen in Figure 4.

- #shift – Immediate shift length
- Sh – Shift type
- opcode – Determines the ALU operation

4	2	1	4	1	4	4	5	2	1	4
cond	00	1	opcode	S	Rn	Rd	#shift	Sh	0	Rm

Figure 4: Data processing instruction example

The Thumb instruction set is an add-on to later generation ARM architectures. It is used to compresses a subset of the ARM instruction set. Thumb is not a compete architecture because not all ARM operations can be compressed into 16-bits. Implementations decompress Thumb instructions into ARM instructions and then they are processed through the pipeline.

ARM does not implement some hardware features found on other processors. The main ARM pipeline has no support for floating point hardware, so a coprocessor is used when full floating point support is needed. ARM also has no hardware support for integer division, so division is performed in software through a series of instructions.

2.3 ARM Implementations and StrongARM SA-110 CPU

From 1983 to 1995 ARM processors used a 3-stage pipeline, after 1995 5-stage pipelines were utilized because of significant increases in performance. The ARM7TDMI CPU core is the current low end ARM processor core. It implements the 3-stage pipeline typically yielding a clock rate of 66Mhz running on a 3.3V power supply reaching 60 MIPS. The ARM9TDMI core utilizes the 5-stage pipeline and significantly increases the performance over the ARM7TDMI core. ARM9TDMI typically runs at 200Mhz with a 2.5V supply voltage and reaches 220 MIPS. Similar to the ARM9TDMI core is the StrongARM SA-110 CPU, the only major difference is the addition of a dedicated adder to calculate branches in the SA-110 rather than utilizing the ALU to calculate the branch address. This makes the SA-110 take one less cycle than the ARM9TDMI when calculating a branch. The SA-110 CPU will now be further discussed because the SimpleScalar/ARM implementation provides very similar performance to the SA-110.

The main features of the SA-110 are:

- a 5-stage pipeline with forwarding
- 16KB 32-way associative instruction cache with 32-byte lines
- 16KB 32-way associative data cache with 32-byte lines

The SA-110 processor core employs a 5-stage pipeline with full forwarding. Because ARM requires a shift operation before the ALU, a barrel shifter was added before the ALU and does not add to the pipeline depth. The five stages of the SA-110 are:

1. Instruction fetch from instruction cache.
2. Instruction decode and register read; branch target calculation and execution.
3. Shift and ALU operation, including data transfer memory address calculation.
4. Data cache access.
5. Result write-back to register file.

These stages are very similar to the ones explained before in the background section. The forwarding paths are values from the ALU result, data loaded from the data cache, and the buffered ALU result. Forwarding is needed because of data dependencies that exist between instructions.

Figure 5 is a diagram of the SA-110 core pipeline. The light green busses represent modifications done to the program counter (PC). The light grey represent the forwarding paths taken to alleviate most data dependencies. Notice the shifter before the ALU, this allows each data processing instruction to be able to shift then perform an ALU operation in one cycle. The adder (+4) located in the execute stage allows multiple register stores and loads. The displacement adder located in the instruction decode allows branch addresses to be calculated during the decode phase without having to calculate the address in the ALU. These are the unique aspects of the SA110 CPU core.

2.4 SimpleScalar/ARM: An ARM Simulator

SimpleScalar has a retargeted version of its simulator for the ARM ISA. It features the ARM7 integer instructions with support for FPA floating point emulation. A cross compiler is also available to compile C code into ARM ELF binaries for simulation. Functional simulation is done by using sim-uop while detailed performance and power estimation is performed by sim-outorder. Included with the SimpleScalar/ARM distribution is a configuration file that configures the SimpleScalar/ARM simulator to perform similarly to a hardware based SA-110 CPU. There is about a 4% difference in performance between SimpleScalar/ARM and a Netwinder SA-110 CPU. Table 3 shows several benchmarks run on the SimpleScalar/ARM simulator and actual Netwinder SA-110 hardware CPU. Data was taken from [6]. The performance of the simulator was very similar to the performance of the hardware CPU.

Benchmark	SimpleScalar/ARM (CPI)	Netwinder SA-110 (CPI)	% Diff
smooth.new	1.02	1.01	0.9
trash.new	22.87	33.7	0.5
ln	1.04	1.02	1.9
ly	1.97	1.91	3.1
bzip2 10	3.20	3.10	3.2
cc1 –O cc1in.i	2.84	2.90	2.1
fft.arm short.pcm	1.45	1.44	0.1

Table 3: SimpleScalar/ARM compared to actual Netwinder SA-110

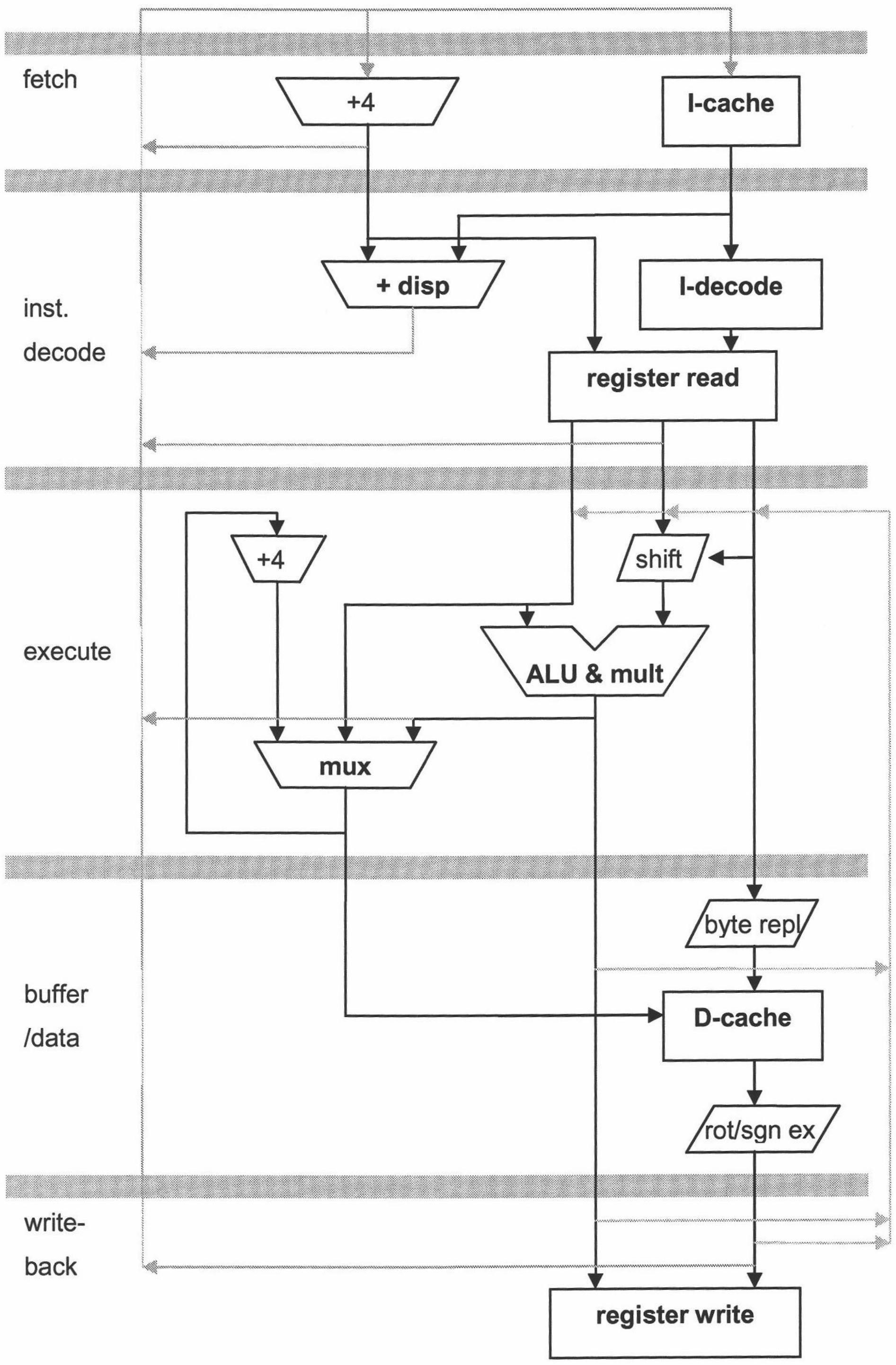

Figure 5: SA-110 core pipeline diagram

3 X32V

3.1 Introduction

X32V is an embedded processor architecture created at Oregon State University. It has several unique features to create dense binaries and increase performance. The main purpose of X32V was to create an architecture that is easily extensible. This means that adding additional functionality or performance enhancements to the processor core would be simple and cost effective. The addition of these enhancements would not be connected through a coprocessor interface, but directly attached to the main pipeline to offer the best performance possible. To make the creation of X32V easier, modules from SimpleScalar were used to complement the X32V processing core. The cache, memory, loader, syscall, and others were taken and lightly modified to work with the X32V architecture.

3.2 X32V Instruction Set

One of the unique features of X32V are the instruction sizes available from the ISA. The three instruction sizes are described below:

- default (32-bit instructions only)
- light (32-bit and 16-bit instructions)
- ultra-light. (32-bit, 24-bit, and 16-bit instructions)

Code density is increased when using light or ultra-light mode, creating smaller binaries, but complexity of the fetch portion of the pipeline is increased. Misalignment of instructions in memory causes the fetch stage to perform multiple fetches to receive an instruction resulting in increased overhead. [5] shows a decrease in executable size of 7% for light mode and 25% for ultra-light for several benchmarks from the MediaBench suite of programs. The performance overhead from using light and ultra-light modes is about 3% additional cycles. Ultra-light mode provides the best option in creating small code size with little overhead.

Table 4 shows the instruction classes and their configuration for 32-bit, 24-bit, and 16-bit instructions. Immediate values are highly influenced by the instruction size because with less bits there is a smaller range for immediate values. The compiler chooses the smallest instruction size possible when creating a binary depending on immediate value ranges. This helps to reduce code size as seen in [5].

32-bit Instruction Format

	4	4	4	4	16			
Load / Store	0000	op1	rd	rs1	disp			
Immediate	0001	op1	rd	rs1	imm			
Branch	0010	op1	rd	rs1	Label			
	4	4	4	4	4	4	4	4
Register	0011	op1	rd	rs1	rs2	op2	op3	op4
	4	4	24					
Jump / Call	0100	op1	label / imm					

24-bit Instruction Format

	4	4	4	4	8	
Load / Store	0101	op1	rd	rs1	disp	
SR Load Store	0101	op1	op2	rs1	disp	
Immediate	0110	op1	rd	rs1	imm	
Branch	0111	op1	rd	rs1	label	
	4	4	4	4	4	4
Register	1000	op1	rd	rs1	rs2	op2
	4	4	16			
Jump / Call	1001	op1	label / imm			

16-bit Instruction Format

Load/Store, Imm, branch		none			
		4	4	4	4
Register	R-1	1010	op1	rd	rs1
	R-2	1011	op1	rd	rs1
		4	4	8	
Jump / Call		1100	op1	label / imm	

Table 4: X32V Instrution set classes

Unique to X32V are several conditional branch instructions described in Table 5. These branches compress a compare and branch into one instruction allowing one cycle execution. The drawback is the additional hardware needed to perform the conditional calculations instead of using the ALU's condition codes. This will likely lead to a longer critical path in the decode stage, where the conditional branches are performed.

A complete listing of all X32V instructions can be found in [5].

B_EQ	Branch if Equal
B_NE	Branch if Not Equal
B_EQZ	Branch if Equal to Zero
B_NEZ	Branch if Not Equal to Zero
B_LTZ	Branch if Less Than Zero
B_GTZ	Branch if Greater Than Zero
B_LTEZ	Branch if Less Than or Equal to Zero
B_GTEZ	Branch if Greater Than or Equal to Zero

Table 5: X32V Conditional Branches

3.3 X32V Architecure

X32V utilizes a classic 5-stage pipeline design similar to many other embedded systems. The stages are listed below:

1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Memory accesses
5. Write back to register file

Figure 6 shows the micro-architectural diagram of the X32V integer pipeline along with an expandable module called EM3. EM3 is a multimedia module that performs single instruction multiple data (SIMD) instructions to significantly increase the performance of multimedia applications like MPEG decompression. To best utilize the architecture, EM3 uses the floating point registers in the FPU, since floating point operations are rarely done in multimedia applications.

One of the unique features of X32V are variable instruction sizes. To handle the different instruction sizes an extra buffer and control logic are needed to correctly fetch instructions from memory. With different instruction sizes, memory is no longer aligned on word boundaries so extra hardware is needed to handle fetching an extra word to retrieve the other portion of an instruction. In default mode, where all instruction are 32-bits, the fetch is a simple process of getting the next instruction in memory because all instruction fall onto word boundaries.

To handle conditional branching additional hardware in the decode stage is needed. A comparator and subtractor are needed to perform the conditional checks before a branch is taken. The target address is also computed in the decode stage, requiring an adder to compute the branch addresses.

The integer pipeline also has a dedicated hardware multiply and divide unit to increase the speed of those operations. Multiply and especially divide operations tend to take more than one cycle to perform resulting in the need to handle multi-cycle instructions.

3.3.1 Handling Multi-cycle Instructions

With multi-cycle instructions, such as multiply and divide, the execute stage takes more than one cycle to compute a result. More cycles are needed because multiply and divide operations, depending on implementation, can take several cycles before a result is computed. Therefore a reservation shift register (RSR) was used to keep track of when an instruction will finish the execute stage. The RSR contains bits that show when an instruction will finish executing and any new instructions will be masked to the RSR to detect if it has the opportunity to be executed. If it cannot execute then it is queued until the RSR shows that there is an open window for it to execute. This solves the problem of out-of-order execution of instructions and keeps everything in-order.

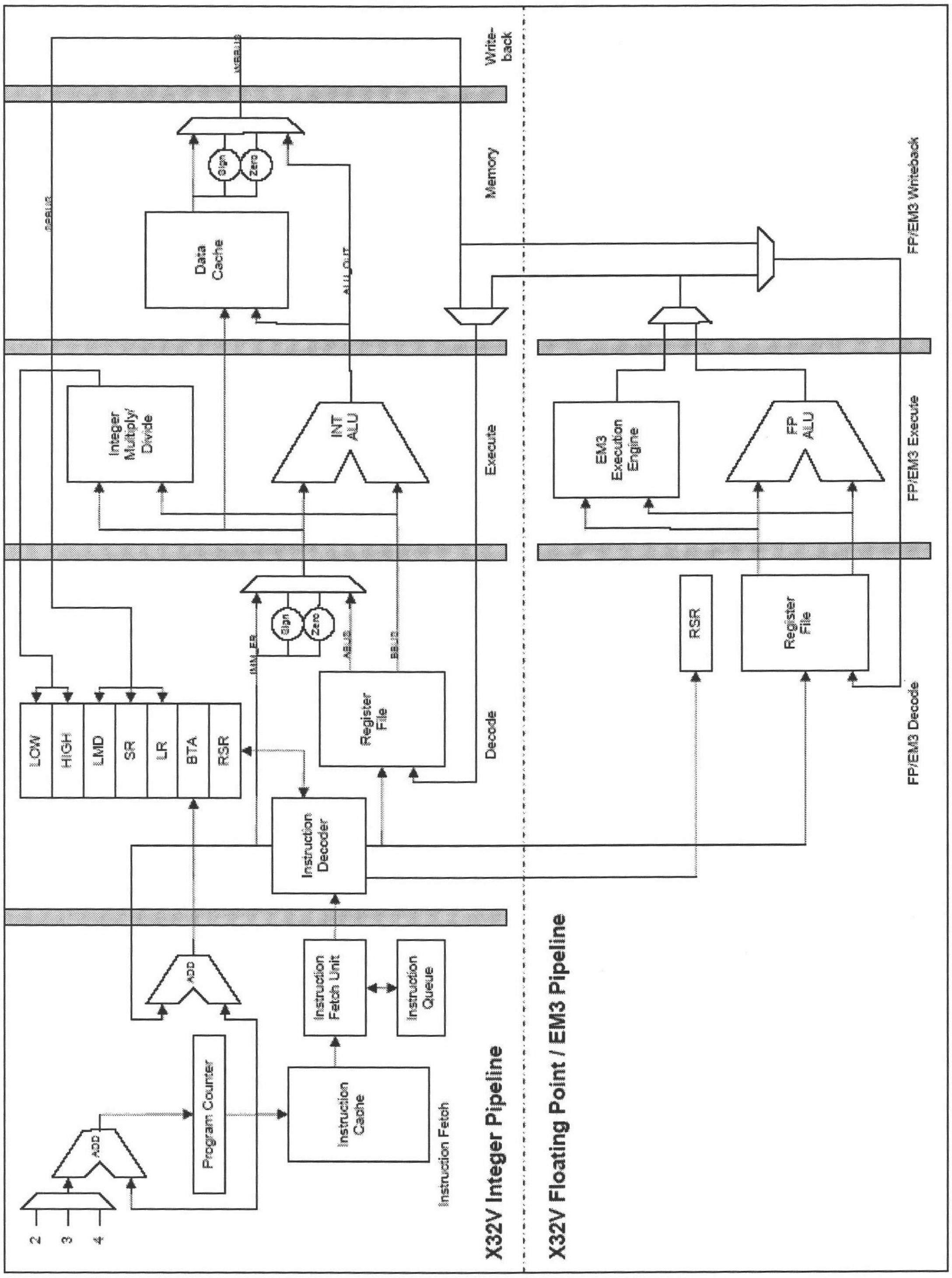

Figure 6: X32V micro-architectural diagram

3.3.2 EM3 Multimedia Module

The EM3 multimedia unit exploits parallel data structures in common multimedia programs to increase performance. For example, common parallel data structures are RGB values stored in sets of three 8-bit integers. Since the ALU is capable of processing 32-bit integers, all three RGB values are able to be computed at the same time. An integer ISA is commonly extended to perform these SIMD operations to process parallel data. Common SIMD extensions are MMX and SSE for Intel and VIS for Sun architectures. Figure 6 shows how EM3 is integrated into the main integer pipeline. EM3 uses the floating point registers to store data. An in depth description of EM3 is seen in [7].

3.3.3 Floating Point Module

The floating point module processes floating point instructions and data. There are two data types supported by the X32V floating point module: single (32-bit) and double (64-bit). There are also instructions to convert between single, double, and integers. Many of the normal floating point operations are supported including multiply, divide, and addition. Figure 6 shows how the floating point module is integrated into the main pipeline. All of the floating point instructions are explained in Appendix A: X32V Floating Point Instructions.

3.4 X32V Simulator

The X32V simulator is an event-driven, cycle accurate simulator, written in C and created for the linux platform. Some of the components from SimpleScalar were used to interface with the X32V core pipeline. Components such as the system call, memory, loader, cache, and statistics modules were used to create the simulator. Figure 7 shows the structure of the X32V simulator. User programs are assembled into binary, conforming to the X32V ISA. The binary is loaded into memory by the loader. Before the simulation begins a configuration file is parsed to determine the memory latency and cache configuration for the simulation. Then the binary is run on the core pipeline and statistics are gathered and outputted to a file when the simulation has finished. The syscall module is a proxy to the operating system when system calls are requested by the running program.

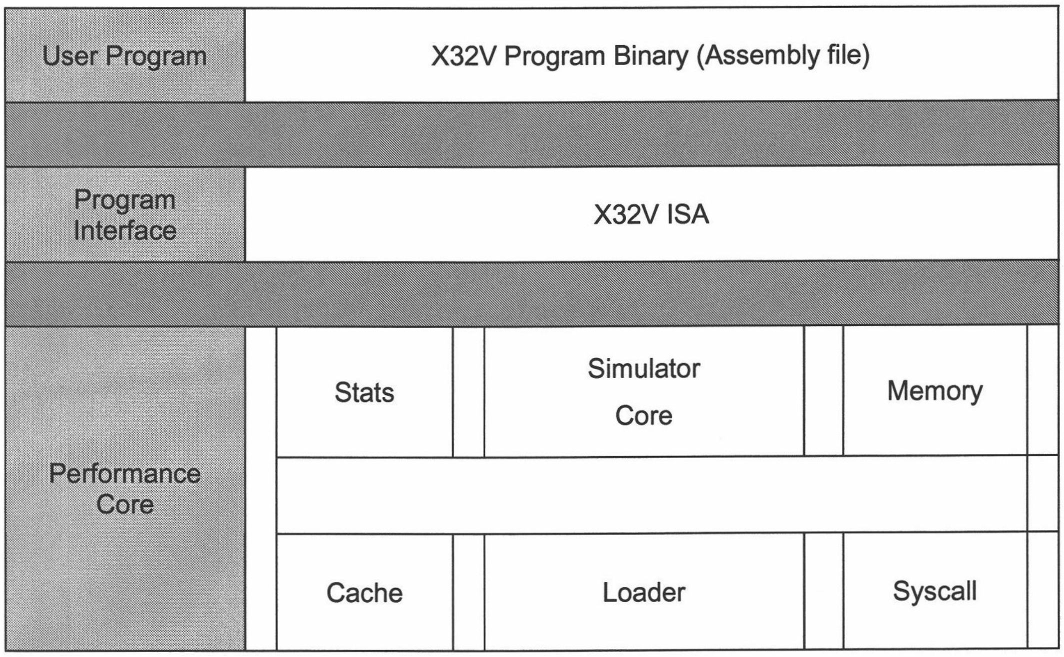

Figure 7: X32V simulator structure

The core pipeline of the simulator is run in reverse order to make it easier to forward data to the different stages. Each function below describes the operation of each stage of the main simulator.

pipe_update()

During a pipe_update all stages of the pipeline are updated. NOP's are entered into the pipeline when memory stalls occur and if there was a taken branch, a one cycle memory stall is needed to fetch the next instruction in instruction memory.

wb_action()

This is where the register file is updated with data from the mem stage.

mem_action()

As the simulator enters this stage, stalls are detected for memory accesses through calls to the cache and memory modules. If there are any stalls in the pipeline, the stage waits until all the stalls are cleared. When all stalls are cleared memory accesses begin only through loads and stores.

ex_action()
Most of the operations done in this stage are arithmetic and involve the ALU or multiply and divide unit. Data processing instructions similar to add, multiply, and shift are executed in this stage.

id_action()
In this stage instruction formats are determined, branch target addresses (BTA) are calculated, and instructions decoded for the next stage. The instruction formats available are default, light, and ultra-light.

if_action()
Instruction are fetched from memory in this function. If a full instruction was not fetched, because of misalignment in memory, then the pipeline is stalled until another memory access can fetch the second part of the instruction.

3.4.1 Modifications to X32V Simulator

Several modifications were made to the original X32V simulator written by John Mark Matson. Instructions were added, the loader was modified, a system call was added, and modifications were made to the assembler. A RSR was added to allow multi-cycle executing instructions. Also validation was done correcting several minor bugs within the simulator.

Multiply and divide instructions were added to the original simulator along with HI and LO registers to hold the result. To make manipulation of the stack as simple as possible PUSH and POP instructions were added. PUSH puts a register value onto the stack and POP takes the last pushed item off the stack. Floating point instruction were added to the original simulator to provide support for hardware based floating point calculations. Several instruction were added to move to and from HI and LO registers, stack pointer, link register, and status register to any general purpose register (GPR).

The loader was modified to accept ELF format binaries. The original simulator only accepted plain binary files. ELF provides features such as dynamic linking and loading, runtime control of programs, and an improved method of providing shared libraries.

There was an additional system call added to create files within an executing program by issuing a system call to the operating system. O_CREAT was added to call the open system call in the UNIX operating system with flags to create a file.

When the X32V simulator was first developed, a compiler had not yet been made. So an assembler was created that converted assembly instructions into its binary representation. The assembler was a PERL script that parsed an assembly file and extracted all necessary information to create the binary equivalent. This is by no means a robust assembler-linker combination, but it provides a relatively simple and highly configurable way to create programs to run on the X32V simulator. To add additional functionality to the assembler, a preprocessor directive was added to preload data memory with values. This simplifies constant values by not having to add them through immediate values in the assembly. The assembler was also modified to include all of the new instructions added to the ISA including the EM3 and floating point instructions.

To allow multi-cycle executing instructions an RSR was added to insure in-order execution. Complementing the RSR was the utilization of several SimpleScalar resources to buffer waiting instructions to be executed.

3.5 Compiler Support for X32V

In collaboration with Electronics and Telecommunications Research Institute (ETRI) of Korea an X32V compiler was developed based on the GNU CC compiler suite. The compiler developed supports compilation only and not assembly and linking. This makes it difficult to run very extensive benchmarks, but allows the comparison of GCC generated programs. Assembly created by GCC for different targets use similar algorithms for code optimizations.

4 Performance Comparison X32V vs. SimpleScalar/ARM (SA-110 Core)

4.1 Introduction

To evaluate X32V's viability as an embedded processor architecture, a comparison must be made between current technologies and the one proposed. The most popular embedded architecture currently is ARM, it is widely used in many applications and a standard in the industry. With many implementations and robust software support, ARM provides the best comparison to X32V. The disadvantage X32V encounters is the lack of an assembler and linker. Without these essential software components intense benchmarks cannot be run. Since the GNU cross compiler is working and producing viable assembly, a smaller benchmark written at Oregon State University by Jarrod Nelson can be used in a comparison. Using the X32V GNU cross-compiler to generate assembly and then using the PERL assembler to convert it straight to binary will result in an executable binary. The compiler used in the SimpleScalar/ARM suite is GNU based and produces similar code to the compiler for X32V. Therefore a comparison between X32V and ARM will be made using the X32V simulator and SimpleScalar/ARM simulator to determine if X32V is a viable embedded processor.

4.2 Benchmark: YCbCr to RGB Conversion

In many multimedia applications color conversion from one color space to another is frequently performed. Especially converting YCbCr to RGB color space in the JPEG and MPEG algorithms. YCbCr is used because information is compacted into a luminance (Y) and chrominance (CbCr) components. The human eye is more sensitive to variations in brightness than color allowing image compression without noticeable changes in image quality. Luminance, being the brightness of an image, should be preserved while sub-sampling can be done on the chrominance portion to compress the image while maintaining good image quality. For example, 8-bits can be used to sample a luminance component while 4-bits are used to sample the chrominance components causing the image to be compressed 33% when compared with the 24-bit RGB version of the image.

The benchmark used in this study converts a YCbCr image into RGB color space. Sub-sampling was done on the chrominance (CbCr) components to compress the image. An RGB image was chosen and converted into its YCbCr color space equivalent then run through the benchmark on the simulator to convert it back to RGB for visual inspection. Using this benchmark provides an easily verifiable result through visual inspection of the final RGB image. The original RGB image converted to YCbCr and the resulting image from the output of the benchmark program should resemble the image in Figure 8.

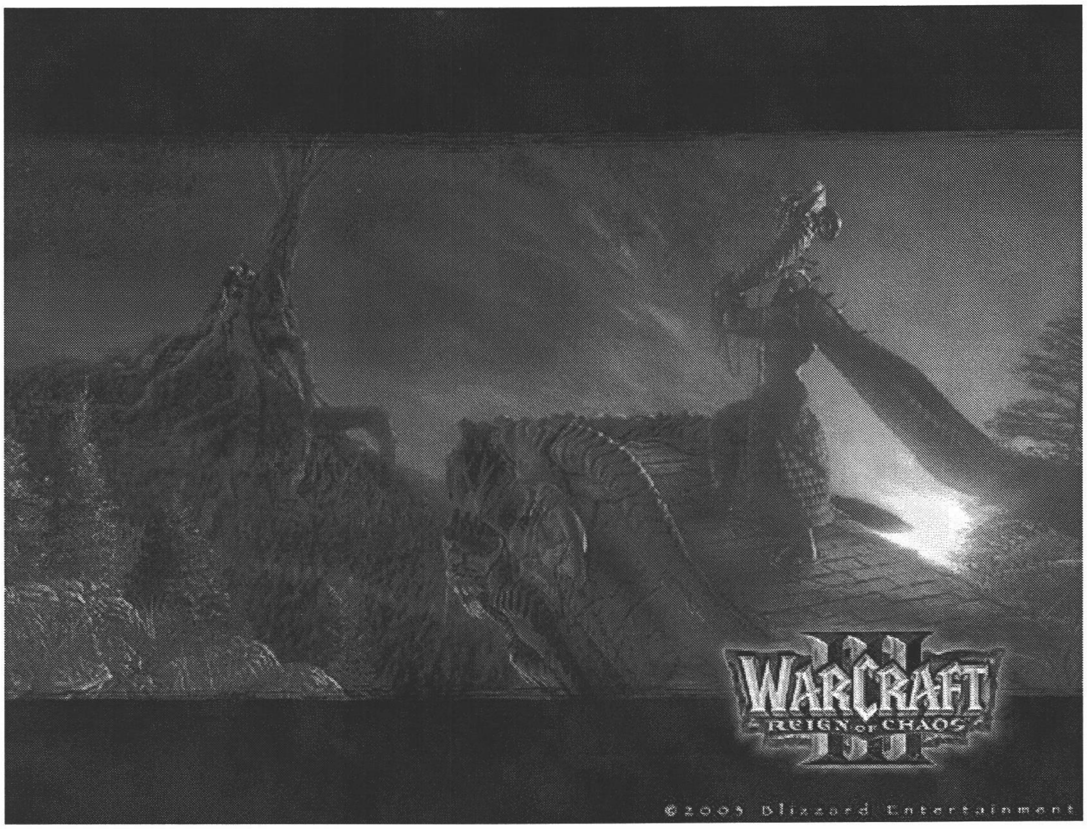

Figure 8: Original image used to convert to YCbCr then converted back to RGB.

4.3 Test Configuration

The benchmark was written in C and compiled using gcc with no optimizations. SimpleScalar/ARM's gcc was able to directly create an executable binary while X32V's gcc was used to compile into assembly and then hand massaged to use with the PERL assembler to create an executable binary. X32V's gcc did not handle system calls, immediate values, or memory mapping correctly. Therefore hand modifications had to be made to correct the assembly to work with the simulator. Default (32-bit instructions) mode was used for all X32V simulations. The simulations were run with a 16KB L1 data cache and 16KB L1 instruction cache with a hit latency of one cycle. The cache replacement policy simulated was a FIFO. Main memory had a 64 cycle latency. Another simulation was done using X32V with the EM3 module to further increase performance. The assembly code for this benchmark was written by hand because the X32V gcc compiler does not support any of the EM3 instructions. The same memory configuration was used during with the EM3 simulation.

4.4 Results

The results of running the YCbCr to RGB conversion benchmark on each processor simulator are shown in Figure 9. To compare different architectures only one statistic is relevant in determining performance and it is how long the processor took to complete a task. Cycle count is how many cycles the benchmark took to complete on the simulated processor. Assuming clock cycle times (CCT) are similar, cycle count tells which architecture provided the best performance (lower numbers are better).

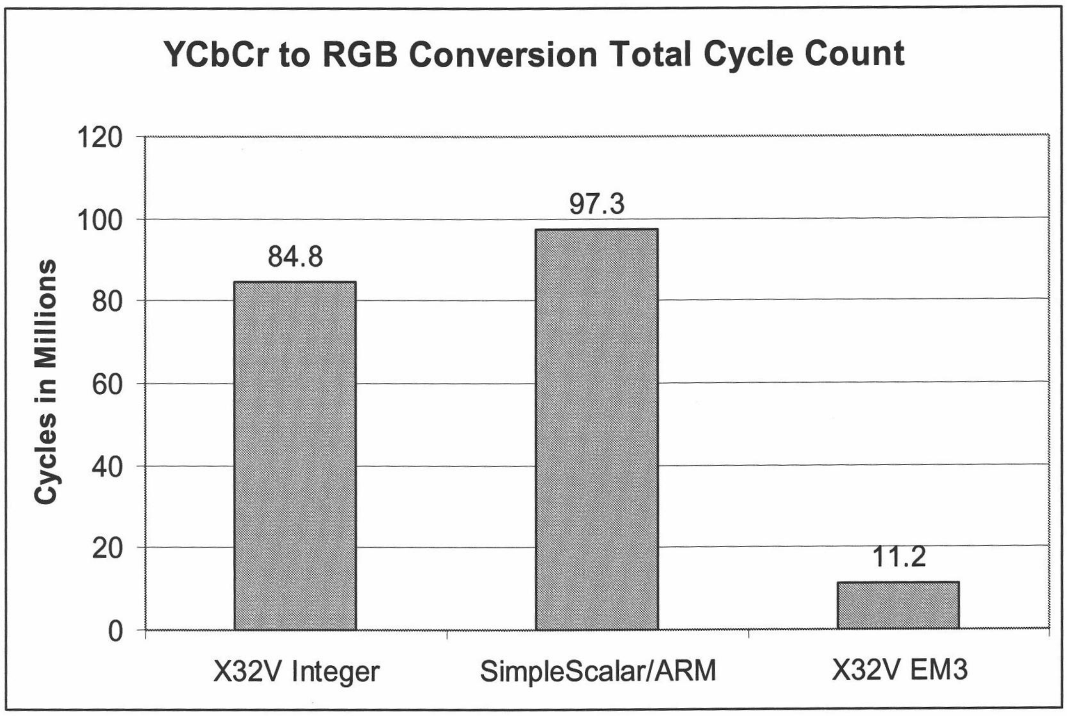

Figure 9: Total cycle count for YCbCr to RGB conversion benchmark

From Figure 9 the best performing architecture between X32V Integer and SimpleScalar/ARM was X32V, a difference of 12.5 million cycles. There are several reasons for this performance increase. In ARM, before a branch is executed a compare is done to set the condition codes allowing the branch to be determined. In X32V a compare and branch can be done in one instruction. Also in X32V, Register R0 is considered to have a value of 0 reducing the number of steps needed to assign a value to a register.

C Code:

```
if (r1<0)
        r1 = 0;
else if (r1 > 255)
        r1 = 255;
```

ARM Assembly:

```
              ldr     r3, [fp, #-28]
              cmp     r3, #0
              bge     .L14
              mov     r3, #0
              str     r3, [fp, #-28]
              b       L15
      L14:    ldr     r3, [fp, #-28]
              cmp     r3, #255
              ble     L15
              mov     r3, #255
              str     r3, [fp, #-28]
      L15:
```

X32V Assembly:

```
              l_w     r2, 28(r15)
              b_gtez  r2, L14
              s_w     r0, 28(r15)
              j_a     L15
      L14:    l_w     r2, 28(r15)
              set_lti r2, r2, 256
              b_ne    r2, r0, L15
              addi    r2, r0, 255
              s_w     r2, 28(r15)
      L15:
```

Figure 10: Example code

Figure 10 shows example code taken from the YCbCr to RGB conversion benchmark. Shown is the ARM assembly representation and the X32V assembly representation of the C code. The code segment is from a portion of the algorithm where saturation checks are performed to make sure that each 8-bit RGB value is between 0 and 255. As stated, two cycles are saved in the X32V implementation because of the integration of conditional branches into one instruction and the use of R0 as a zero register. In the ARM implementation, line 2 does a compare before a branch and in line 4 a move is used to set a register to zero before storing it in memory. In the X32V implementation lines 2 and 3 do what takes ARM four instructions. The compare and branch is done in one instruction while the clearing of registers does not need to be done because R0 is the zero register. Saving two cycles per RGB value results in 6 cycles saved per pixel. In a 640x480 sized image there are 307,200 pixels resulting in an overall savings of 1,843,200 cycles (6 X 307,200) for the X32V implementation.

The conditional branches and zero register do not account for all of the 12.5 million cycle difference. There were minor differences in code between the SimpleScalar/ARM compiler and the X32V compiler that accounted for most of the 12.5 million cycles. In the ARM implementation a move and negate and three subtracts were executed before most loads. This was done to calculate the address of where the load was going to fetch data from memory. X32V used a more optimized method of using a direct immediate value rather than calculating an address every time before a load. The gcc version for the SimpleScalar compiler is 2.95.2 while X32V's compiler version is 3.3. The difference in these compiler versions accounts for the optimization seen in X32V.

Table 6 shows an instruction profile of the benchmark for each simulator. ARM's instruction profile shows a large amount of subtracts being performed. This is a result of the three additional subtracts before a load. Most of the time both processors were performing integer ALU operations. Having an additional ALU would greatly increase the performance of this benchmark.

X32V with the addition of the EM3 module significantly increase the performance of the benchmark program. Figure 9 shows approximately an 8 fold decrease in cycles when compared with ARM or X32V integer pipelines. EM3 utilizes a SIMD format that is perfect for RGB calculation. Each RGB value can be processed in once cycle rather than performing calculations on each R, each G, and each B value as in normal integer pipelines. Also the assembly that was generated for EM3 was written by hand and highly optimized.

Instruction Profile: ARM		
	# Inst Executed	Percent of Total
Total	**80907317**	**100.00%**
loads	13225840	16.35%
store	5256233	6.50%
uncond branch	317921	0.39%
cond branch	3065359	3.79%
int comp	59041937	72.97%
b	3383037	4.18%
rsb	1229849	1.52%
add	9318027	11.39%
mov	15062814	18.62%
sub	20902411	25.84%
add	2171730	2.68%
mvn	7069967	8.74%
b	319369	0.39%
cmp	3065240	3.79%
str	5254338	6.50%
ldr	13222297	16.35%

Instruction Profile: X32V		
	#Inst Executed	Percent of Total
Total	**62126024**	**100.00%**
loads	14453945	23.27%
store	5870177	9.45%
uncond branch	317592	0.51%
cond branch	3371691	5.43%
int comp	38112599	61.35%
l_w	12610745	20.30%
l_b	1843200	2.97%
s_w	4948577	7.97%
s_b	921600	1.48%
lg_ori	1843209	2.97%
addi	308111	0.50%
add_ui	3091684	4.98%
sub_ui	921600	1.48%
sft_lli	7681921	12.37%
sft_rli	1536480	2.47%
sft_rai	3073920	4.95%
set_lti	1220332	1.96%
mov_upi	1843209	2.97%
lg_xor	2151319	3.46%
add_u	10752967	17.31%
sub_u	1229761	1.98%
mov	2458086	3.96%
b_ne	2141931	3.45%
b_gtez	1229760	1.98%
j_a	317592	0.51%

Table 6: Instruction profile

5 Future Work and Conclusion

5.1 Future Work

The development of X32V is an ongoing process and requires much work to make it an established processor architecture. The most important work needs to be done on the integration of an assembler and linker into X32V's compiler. This will allow more advanced benchmarks to be run on the simulator. Benchmarks such as MediaBench, SPEC, and DhryStone. Porting the gcc version to a similar version used in SimpleScalar/ARM would be essential to create a level playing field for architectural comparison.

Integrating the Wattch power simulation tool into X32V would lead to interesting studies on power consumption of X32V.

Creating HDL code and utilizing it in an FPGA as a soft coded processor would validate the hardware and provide delay information for the different stages. Using HDL to synthesize a core would determine the maximum cycle time for a certain process and the amount of power the processor will consume. With SimpleScalar/ARM using the SA-110 configuration, performance of X32V can be extrapolated to perform closely to SA-110 performance. Having an HDL description will validate this prediction.

5.2 Conclusion

X32V presents a new and configurable embedded processor architecture. Modules such as EM3 and floating point integrate with the main X32V pipeline to create a high performance pipeline with out the overhead of a co-processor interface. When compared to an established processor such as ARM, X32V shows promise by performing similarly to ARM. Unique features such as light and ultra-light modes, configurable modules, and integrated conditional branching help X32V differentiate itself from other architectures. With unique features and performance similar to ARM, X32V is a viable processor architecture.

Bibliography

[1] Furber, S., ARM System-on-Chip Architecture, Second ed. , Great Britain: Pearson Education Limited, 2000.

[2] Hennessy, J. L. and Patterson, D. A., Computer Architecture A Quantitative Approach, Third ed. , California: Morgan Kaufmann Publishers, 2003.

[3] Austin, T. M., Larson, E., and Ernst, D., " SimpleScalar: An Infrastructure for Computer System Modeling," IEEE Computer, pp. 59-67, February 2002.

[4] Burger, D. and Austin, T. M., "The SimpleScalar Tool Set, Version 2.0," available at www.simplescalar.com

[5] Matson, J. M., "Designing a Reconfigurable Embedded Processor", MS Thesis, Oregon State University, 2003.

[6] Austin, T. M., README.ARM, SimpleScalar/ARM, University of Michigan, 2003.

[7] Zier, D., "Designing Multimedia Extensions for Configurable Processors", MS Thesis, Oregon State University, 2004.

[8] Zier, D., et. al., "X32V: A Design of a Configurable Processor Core for Embedded Systems", International Conference on Embedded Systems and Applications, Las Vegas, 2004.

[9] Tensilica, "Xtensa Architecture and Performance", White Paper, Tensilica, Inc., Santa Clara, CA, 2002.

[10] Lee, C., Potkonjak M., and Mangione-Smith, W. H., "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", University of California, Los Angeles, 1998.

[11] Wall, L., Christiansen, T., and Schwartz, R. L., Programming Perl, Second ed., California: O'Reilly and Associates Inc., 1996.

[12] Haviland, K., Gray D., and Salama, B., UNIX System Programming, Second ed., Great Britain: Pearson Education Limited, 1998.

Appendix A: X32V Floating Point Instructions

Introduction

The following pages outline each instruction in the X32V Floating Point ISA. Detailed information about instruction type, format, usage, and encoding are given.

Symbol Definition

=	Substitute left side of operator with right side
+	Addition
-	Subtraction
*	Multiplication
/	Division
= =	Test equality
! =	Test inequality
>	Greater Than
<	Less Than
&	Bit wise Logical AND
|	Bit wise Logical OR
^	Bit wise Logical XOR
||	Join or Concatenate
<<	Bit wise Shift Left
>>	Bit wise Shift Right
rs1	Source register one
rs2	Source register two
rd	Destination register
MEM(0x2a)	Value at main memory address 0x2a
$`0'^{8}$	Zero extended 8 places
$`imm_{15}'^{16}$	15th bit of immediate value, sign extended 16 places

Notes on Opcode Implementation

The first opcode field that signifies the mode of the instruction is the same as the EM3 opcode:

- 32-bit FP = 32-bit EM3 = 13 = 1101_2
- 24-bit FP = 24-bit EM3 = 14 = 1110_2

- 16-bit FP is not support due to the need for a third opcode byte. This is similar to the EM3 instruction set.

The value in the second opcode field signifies the precision of the floating-point operation:

- Single – 1111_2
- Double – 1110_2

The third opcode field determines the particular floating point operation to be performed. There are 16 individual floating operations and 2 different precision formats for a total of 32 floating point operations.

ADD_S Single Floating Point Addition

Description:

The contents of FPR rs1 and FPR rs2 are arithmetically added. The result of the operation (in single precision format) is placed in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

ADD_S rd, rs1, rs2

Operation:

32-bit / 24-bit

rd = rs1 + rs2

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	rs2	0000	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	rs2	0000
4	4	4	4	4	4

SUB_S

Single Floating Point Subtraction

Description:

The contents of FPR rs1 and FPR rs2 are arithmetically subtracted. The result of the operation (in single precision format) is placed in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

SUB_S rd, rs1, rs2

Operation:

32-bit / 24-bit

rd = rs1 – rs2

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	rs2	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	rs2	0001
4	4	4	4	4	4

MUL_S

Single Floating Point Multiplication

Description:

The contents of FPR rs1 and FPR rs2 are arithmetically mutliplicated. The result of the operation (in single precision format) is placed in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

MUL_S rd, rs1, rs2

Operation:

32-bit / 24-bit

rd = rs1 * rs2

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	rs2	0010	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	rs2	0010
4	4	4	4	4	4

DIV_S Single Floating Point Division

Description:

The contents of FPR rs1 and FPR rs2 are arithmetically divided. The result of the operation (in single precision format) is placed in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

DIV_S rd, rs1, rs2

Operation:

32-bit / 24-bit

rd = rs1 / rs2

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	rs2	0011	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	rs2	0011
4	4	4	4	4	4

CP_EQ_S Single Compare Equal

Description:

Compares the contents of FPR rs1 and FPR rs2 as single precision floating-point numbers. If rs1 equals rs2 then GPR rd is set to 1, otherwise it is set to 0. Pay careful attention to this instruction and note that rd is an integer register and not a floating-point register.

Type:

Floating Point

Format:

32-bit / 24-bit

CP_EQ_S rd, rs1, rs2

Operation:

32-bit / 24-bit

```
if( rs1 == rs2 )
    rd = 1
else
    rd = 0
```

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	rs2	0100	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	rs2	0100
4	4	4	4	4	4

CP_LE_S

Single Compare Less Than Equal

Description:

Compares the contents of FPR rs1 and FPR rs2 as single precision floating-point numbers. If rs1 is less than or equal to rs2 then GPR rd is set to 1, otherwise it is set to 0. Pay careful attention to this instruction and note that rd is an integer register and not a floating-point register.

Type:

Floating Point

Format:

32-bit / 24-bit

CP_LE_S rd, rs1, rs2

Operation:

32-bit / 24-bit

```
if( rs1 <= rs2 )
    rd = 1
else
    rd = 0
```

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	rs2	0101	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	rs2	0101
4	4	4	4	4	4

CP_LT_S Single Compare Less Than

Description:

Compares the contents of FPR rs1 and FPR rs2 as single precision floating-point numbers. If rs1 is less than rs2 then GPR rd is set to 1, otherwise it is set to 0. Pay careful attention to this instruction and note that rd is an integer register and not a floating-point register.

Type:

Floating Point

Format:

32-bit / 24-bit

CP_LT_S rd, rs1, rs2

Operation:

32-bit / 24-bit

```
if( rs1 < rs2 )
    rd = 1
else
    rd = 0
```

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	rs2	0110	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	rs2	0110
4	4	4	4	4	4

CVT_S_D Convert Single To Double

Description:

Converts the contents of FPR rs1 (as single precision floating-point numbers) to a double precision floating-point number. The resulting double precision element will be placed in FPR rd. FPR rd must be an even number element to prevent double misalignment in the register file.

Type:

Floating Point

Format:

32-bit / 24-bit

CVT_S_D rd, rs1

Operation:

32-bit / 24-bit

rd_D = CVT($rs1_S$, double)

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	unused	0111	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	unused	0111
4	4	4	4	4	4

CVT_S_W Convert Single To Integer Word

Description:

Converts the contents of FPR rs1 (as single precision floating-point numbers) to an integer word. The resulting integer word will be placed in GPR rd. Pay careful attention to this instruction and note that rd is an integer register and not a floating-point register.

Type:

Floating Point

Format:

32-bit / 24-bit

CVT_S_W rd, rs1

Operation:

32-bit / 24-bit

rd_W = CVT($rs1_S$, word)

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	unused	1000	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	unused	1000
4	4	4	4	4	4

CVT_W_S

Convert Integer Word To Single

Description:

Converts the contents of GPR rs1 (as an integer word) to a single precision floating-point number. The resulting single floating-point number will be placed in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

CVT_W_S rd, rs1

Operation:

32-bit / 24-bit

rd_S = CVT($rs1_W$, single)

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	unused	1001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	unused	1001
4	4	4	4	4	4

L_S Load Single

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The single floating-point number in memory at this address is copied into FPR rd.

Type:

Floating Point Load / Store

Format:

32-bit / 24-bit

L_S rd, disp(rs1)

Operation:

32-bit

rd = MEM (rs1 + ($'\text{disp}_{11}'^{20}$ || disp))

24-bit

rd = MEM (rs1 + ($'\text{disp}_{3}'^{28}$ || disp))

Encoding:

31 28	27 24	23 20	19 16	15 12	11 0
32-bit FP	1111	rd	rs1	1011	displacement
4	4	4	4	4	12

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	1011	disp
4	4	4	4	4	4

S_S Store Single

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The single floating-point number in FPR rd is stored in memory at this address.

Type:

Floating Point Load / Store

Format:

32-bit / 24-bit

S_S rd, disp(rs1)

Operation:

32-bit

MEM (rs1 + ($\text{‘disp}_{11}\text{’}^{20}$ || disp)) = rd

24-bit

MEM (rs1 + ($\text{‘disp}_{3}\text{’}^{28}$ || disp)) = rd

Encoding:

31 28	27 24	23 20	19 16	15 12	11 0
32-bit FP	1111	rd	rs1	1100	displacement
4	4	4	4	4	12

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	1100	disp
4	4	4	4	4	4

MOV_S

Move Single Floating Point

Description:

The contents of FPR rs1 are moved to FPR rd. The move is done as interpreting the results as a single precision floating-point number.

Type:

Floating Point

Format:

32-bit / 24-bit

MOV_S rd, rs1

Operation:

32-bit / 24-bit

rd = rs1

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	unused	1101	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	unused	1101
4	4	4	4	4	4

ABS_S

Single Floating Point Absolute Value

Description:

Computes the absolute value of the floating-point single in FPR rs1. The result is stored in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

ABS_S rd, rs1

Operation:

32-bit / 24-bit

rd = | rs1 |

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	unused	1110	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	unused	1110
4	4	4	4	4	4

NEG_S

Negate Single

Description:

Negate the floating-point single in register FPR rs1 and put it in register FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

NEG_S rd, rs1

Operation:

32-bit / 24-bit

rd = -rs1

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1111	rd	rs1	unused	1111	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1111	rd	rs1	unused	1111
4	4	4	4	4	4

ADD_D Double Floating Point Addition

Description:

The contents of FPR rs1 and FPR rs2 are arithmetically added. The result of the operation (in double precision format) is placed in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

ADD_D rd, rs1, rs2

Operation:

32-bit / 24-bit

rd = rs1 + rs2

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	rs2	0000	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	rs2	0000
4	4	4	4	4	4

SUB_D

Double Floating Point Subtraction

Description:

The contents of FPR rs1 and FPR rs2 are arithmetically subtracted. The result of the operation (in double precision format) is placed in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

SUB_D rd, rs1, rs2

Operation:

32-bit / 24-bit

rd = rs1 – rs2

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	rs2	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	rs2	0001
4	4	4	4	4	4

MUL_D

Single Floating Point Multiplication

Description:

The contents of FPR rs1 and FPR rs2 are arithmetically multiplied. The result of the operation (in double precision format) is placed in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

MUL_D rd, rs1, rs2

Operation:

32-bit / 24-bit

rd = rs1 * rs2

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	rs2	0010	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	rs2	0010
4	4	4	4	4	4

DIV_D Double Floating Point Division

Description:

The contents of FPR rs1 and FPR rs2 are arithmetically divided. The result of the operation (in double precision format) is placed in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

DIV_D rd, rs1, rs2

Operation:

32-bit / 24-bit

rd = rs1 / rs2

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	rs2	0011	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	rs2	0011
4	4	4	4	4	4

CP_EQ_D Double Compare Equal

Description:

Compares the contents of FPR rs1 and FPR rs2 as double precision floating-point numbers. If rs1 equals rs2 then GPR rd is set to 1, otherwise it is set to 0. Pay careful attention to this instruction and note that rd is an integer register and not a floating-point register.

Type:

Floating Point

Format:

32-bit / 24-bit

CP_EQ_D rd, rs1, rs2

Operation:

32-bit / 24-bit

```
if( rs1 == rs2 )
    rd = 1
else
    rd = 0
```

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	rs2	0100	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	rs2	0100
4	4	4	4	4	4

CP_LE_D

Double Compare Less Than Equal

Description:

Compares the contents of FPR rs1 and FPR rs2 as double precision floating-point numbers. If rs1 is less than or equal to rs2 then GPR rd is set to 1, otherwise it is set to 0. Pay careful attention to this instruction and note that rd is an integer register and not a floating-point register.

Type:

Floating Point

Format:

32-bit / 24-bit

CP_LE_D rd, rs1, rs2

Operation:

32-bit / 24-bit

```
if( rs1 <= rs2 )
    rd = 1
else
    rd = 0
```

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	rs2	0101	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	rs2	0101
4	4	4	4	4	4

CP_LT_D Double Compare Less Than

Description:

Compares the contents of FPR rs1 and FPR rs2 as double precision floating-point numbers. If rs1 is less than rs2 then GPR rd is set to 1, otherwise it is set to 0. Pay careful attention to this instruction and note that rd is an integer register and not a floating-point register.

Type:

Floating Point

Format:

32-bit / 24-bit

CP_LT_D rd, rs1, rs2

Operation:

32-bit / 24-bit

```
if( rs1 < rs2 )
    rd = 1
else
    rd = 0
```

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	rs2	0110	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	rs2	0110
4	4	4	4	4	4

CVT_D_S

Convert Double To Single

Description:

Converts the contents of FPR rs1 (as double precision floating-point numbers) to a single precision floating-point number. The resulting double precision element will be placed in FPR rd. FPR rd must be an even number element to prevent double misalignment in the register file.

Type:

Floating Point

Format:

32-bit / 24-bit

CVT_D_S rd, rs1

Operation:

32-bit / 24-bit

rd_S = CVT($rs1_D$, single)

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	unused	0111	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	unused	0111
4	4	4	4	4	4

CVT_D_W — Convert Double To Integer Word

Description:

Converts the contents of FPR rs1 (as double precision floating-point numbers) to an integer word. The resulting integer word will be placed in GPR rd. Pay careful attention to this instruction and note that rd is an integer register and not a floating-point register.

Type:

Floating Point

Format:

32-bit / 24-bit

CVT_D_W rd, rs1

Operation:

32-bit / 24-bit

rd_W = CVT($rs1_D$, word)

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	unused	1000	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	unused	1000
4	4	4	4	4	4

CVT_W_D

Convert Integer Word To Double

Description:

Converts the contents of GPR rs1 (as an integer word) to a double precision floating-point number. The resulting single floating-point number will be placed in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

CVT_W_D rd, rs1

Operation:

32-bit / 24-bit

rd_D = CVT($rs1_W$, double)

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	unused	1001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	unused	1001
4	4	4	4	4	4

L_D Load Double

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The double floating-point number in memory at this address is copied into FPR rd.

Type:

Floating Point Load / Store

Format:

32-bit / 24-bit

L_D rd, disp(rs1)

Operation:

32-bit

$rd = MEM\ (rs1 + (\text{'}disp_{11}\text{'}^{20}\ ||\ disp))$

24-bit

$rd = MEM\ (rs1 + (\text{'}disp_{3}\text{'}^{28}\ ||\ disp))$

Encoding:

31 28	27 24	23 20	19 16	15 12	11 0
32-bit FP	1110	rd	rs1	1011	displacement
4	4	4	4	4	12

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	1011	disp
4	4	4	4	4	4

S_D

Store Double

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The double floating-point number in FPR rd is stored in memory at this address.

Type:

Floating Point Load / Store

Format:

32-bit / 24-bit

S_D rd, disp(rs1)

Operation:

32-bit	*24-bit*
MEM (rs1 + ('disp_{11}'^{20} || disp)) = rd	MEM (rs1 + ('disp_3'^{28} || disp)) = rd

Encoding:

31 28	27 24	23 20	19 16	15 12	11 0
32-bit FP	1110	rd	rs1	1100	displacement
4	4	4	4	4	12

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	1100	disp
4	4	4	4	4	4

MOV_D

Move Double Floating Point

Description:

The contents of FPR rs1 are moved to FPR rd. The move is done as interpreting the results as a double precision floating-point number.

Type:

Floating Point

Format:

32-bit / 24-bit

MOV_D rd, rs1

Operation:

32-bit / 24-bit

rd = rs1

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	unused	1101	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	unused	1101
4	4	4	4	4	4

ABS_D

Double Floating Point Absolute Value

Description:

Computes the absolute value of the floating-point double in FPR rs1. The result is stored in FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

ABS_D rd, rs1

Operation:

32-bit / 24-bit

rd = | rs1 |

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	unused	1110	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	unused	1110
4	4	4	4	4	4

NEG_D

Negate Double

Description:

Negate the floating-point double in register FPR rs1 and put it in register FPR rd.

Type:

Floating Point

Format:

32-bit / 24-bit

NEG_D rd, rs1

Operation:

32-bit / 24-bit

rd = -rs1

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit FP	1110	rd	rs1	unused	1111	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit FP	1110	rd	rs1	unused	1111
4	4	4	4	4	4

Appendix B: Additional X32V Integer Instructions added to original specification

MOV_FHI Move from HI Register

Description:

The contents of HI are copied to GPR rd. The content of HI does not change.

Type:

Register

Format:

<u>*32-bit / 24-bit / 16-bit*</u>

MOV_FHI rd

Operation:

<u>*32-bit / 24-bit / 16-bit*</u>

rd = hi

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1010	rd	unused	unused	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1010	rd	unused	unused	0001
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	1010	rd	unused
4	4	4	4

MOV_FLO

Move from LO Register

Description:

The contents of LO are copied to GPR rd. The content of LO does not change.

Type:

Register

Format:

32-bit / 24-bit / 16-bit

MOV_FLO rd

Operation:

32-bit / 24-bit / 16-bit

rd = lo

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1011	rd	unused	unused	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1011	rd	unused	unused	0001
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	1011	rd	unused
4	4	4	4

MOV_THI

Move to HI Register

Description:

The contents of GPR rs are copied to HI. The content of GPR rs does not change.

Type:

Register

Format:

32-bit / 24-bit / 16-bit

MOV_THI rs

Operation:

32-bit / 24-bit / 16-bit

hi = rs

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1100	rs	unused	unused	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1100	rs	unused	unused	0001
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	1100	rs	unused
4	4	4	4

MOV_TLO Move to LO Register

Description:

The contents of GPR rs are copied to LO. The content of GPR rs does not change.

Type:

Register

Format:

32-bit / 24-bit / 16-bit

MOV_TLO rs

Operation:

32-bit / 24-bit / 16-bit

lo = rs

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1101	rs	unused	unused	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1101	rs	unused	unused	0001
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	1101	rs	unused
4	4	4	4

MOV_FSR Move from Status Register

Description:

The contents of SR are copied to GPR rd. The content of SR does not change.

Type:

Register

Format:

32-bit / 24-bit / 16-bit

MOV_FSR rd

Operation:

32-bit / 24-bit / 16-bit

rd = sr

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1000	rd	unused	unused	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1000	rd	unused	unused	0001
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	1000	rd	unused
4	4	4	4

MOV_TSR Move to Status Register

Description:

The contents of GPR rs are copied to SR. The content of GPR rs does not change.

Type:

Register

Format:

32-bit / 24-bit / 16-bit

MOV_TSR rs

Operation:

32-bit / 24-bit / 16-bit

sr = rs

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1001	rs	unused	unused	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1001	rs	unused	unused	0001
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	1001	rs	unused
4	4	4	4

MOV_FSP Move from Stack Pointer

Description:

The contents of GPR rs are copied from SP. The content of GPR rs does not change.

Type:

Register

Format:

32-bit / 24-bit / 16-bit

MOV_FSP rs

Operation:

32-bit / 24-bit / 16-bit

sp = rs

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1110	rs	unused	unused	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1110	rs	unused	unused	0001
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	1110	rs	unused
4	4	4	4

MOV_TSP

Move to Stack Pointer

Description:

The contents of GPR rs are copied to SP. The content of GPR rs does not change.

Type:

Register

Format:

<u>*32-bit / 24-bit / 16-bit*</u>

MOV_TSP rs

Operation:

<u>*32-bit / 24-bit / 16-bit*</u>

sp = rs

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1111	rs	unused	unused	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1111	rs	unused	unused	0001
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	1111	rs	unused
4	4	4	4

MOV_FLR

Move from Link Register

Description:

The contents of GPR rs are copied from LR. The content of GPR rs does not change.

Type:

Register

Format:

32-bit / 24-bit / 16-bit

MOV_FLR rs

Operation:

32-bit / 24-bit / 16-bit

lr = rs

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	0000	rs	unused	unused	0010	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	0000	rs	unused	unused	0010
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	0000	rs	unused
4	4	4	4

MOV_TLR Move to Link Register

Description:

The contents of GPR rs are copied to LR. The content of GPR rs does not change.

Type:

Register

Format:

32-bit / 24-bit / 16-bit

MOV_TLR rs

Operation:

32-bit / 24-bit / 16-bit

lr = rs

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	0001	rs	unused	unused	0010	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	0001	rs	unused	unused	0010
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	0001	rs	unused
4	4	4	4

L_SR Load Status Register

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. Then the contents at the calculated address are loaded into the Status Register.

Type:

Load / Store

Format:

32-bit / 24-bit

L_SR disp(rs1)

Operation:

32-bit

sr = MEM (rs1 + ('disp_{15}'^{16} || disp))

24-bit

sr = MEM (rs1 + ('disp_{7}'^{24} || disp))

Encoding:

31 28	27 24	23 20	19 16	15 0
32-bit L/S	1000	unused	rs1	displacement
4	4	4	4	16

23 20	19 16	15 12	11 8	7 0
24-bit L/S	1000	unused	rs1	displacement
4	4	4	4	8

S_SR Store Status Register

Description:

The contents of GPR rs1 are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. Then the contents of the Status Register are loaded into the calculated address.

Type:

Load / Store

Format:

32-bit / 24-bit

S_SR disp(rs1)

Operation:

32-bit

MEM (rs1 + (‘disp$_{15}$’^{16} || disp)) = sr

24-bit

MEM (rs1 + (‘disp$_{7}$’^{24} || disp)) = sr

Encoding:

31 28	27 24	23 20	19 16	15 0
32-bit L/S	1001	unused	rs1	displacement
4	4	4	4	16

23 20	19 16	15 12	11 8	7 0
24-bit L/S	1001	unused	rs1	displacement
4	4	4	4	8

PUSH

Push onto Stack

Description:

The register value rd is placed onto the next open position in the stack. Then the stack pointer is decremented by 4. The stack pointer initially begins at 0xFFFFFFF8.

Type:

Register

Format:

32-bit / 24-bit / 16-bit

PUSH rd

Operation:

32-bit / 24-bit / 16-bit

MEM[SP] = rd

SP = SP - 4

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1100	unused	rs	unused	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1100	unused	rs	unused	0001
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	1100	unused	rs
4	4	4	4

POP

Pop from Stack

Description:

First the stack pointer is incremented by 4. Then register value rd is placed onto the next open position in the stack. The stack pointer initially begins at 0xFFFFFFF8.

Type:

Register

Format:

32-bit / 24-bit / 16-bit

PUSH rd

Operation:

32-bit / 24-bit / 16-bit

rd = MEM[SP]

SP = SP + 4

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1101	unused	rd	unused	0001	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1101	unused	rd	unused	0001
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-2	1101	unused	rd
4	4	4	4

MUL Signed Integer Multiplication

Description:

The contents of GPR rs1 are multiplied by the contents of GPR rs2 using two's complement format. The least significant word of the result is placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type:

Register

Format:

32-bit / 24-bit	*16-bit*
MUL rd, rs1, rs2	MUL rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = (rs1 * rs2)$_{31-0}$	rd = (rs1 * rs2)$_{31-0}$

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1100	rd	rs1	rs2	0000	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1100	rd	rs1	rs2	0000
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-1	1100	rd	rs
4	4	4	4

MUL_U

Unsigned Integer Multiplication

Description:

The contents of GPR rs1 are multiplied by the contents of GPR rs2. Register contents are all positive values. The least significant word of the result is placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type:

Register

Format:

32-bit / 24-bit	*16-bit*
MUL_U rd, rs1, rs2	MUL_U rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 * rs2	rd = rd * rs

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1101	rd	rs1	rs2	0000	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1101	rd	rs1	rs2	0000
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-1	1101	rd	rs
4	4	4	4

DIV

Signed Integer Division

Description:

The contents of GPR rs1 are divided by the contents of GPR rs2 using two's complement format. The quotient is rounded toward zero and placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type:

Register

Format:

32-bit / 24-bit	*16-bit*
DIV rd, rs1, rs2	DIV rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 / rs2	rd = rd / rs

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1110	rd	rs1	rs2	0000	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1110	rd	rs1	rs2	0000
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-1	1110	rd	rs
4	4	4	4

DIV_U Unsigned Integer Division

Description:

The contents of GPR rs1 are divided by the contents of GPR rs2. Register contents are all treated as positive integers. The quotient is rounded toward zero and placed in GPR rd. When using a 16-bit format, the contents of GPR rd are used as the first source register.

Type:

Register

Format:

32-bit / 24-bit	*16-bit*
DIV_U rd, rs1, rs2	DIV_U rd, rs

Operation:

32-bit / 24-bit	*16-bit*
rd = rs1 / rs2	rd = rd / rs

Encoding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
32-bit R	1111	rd	rs1	rs2	0000	unused	unused
4	4	4	4	4	4	4	4

23 20	19 16	15 12	11 8	7 4	3 0
24-bit R	1111	rd	rs1	rs2	0000
4	4	4	4	4	4

15 12	11 8	7 4	3 0
16-bit R-1	1111	rd	rs
4	4	4	4