CompArch Final Project: Digital Signal Processing

Vivien Chen, Lauren Pudvan, Samantha Young

December 12, 2018

1 Abstract

For this project, we studied digital signal processing (DSP). The digital signal process takes analog signals like audio, voice, video, temperature, or pressure that have already been digitized and then manipulates them mathematically. DSP is characterized by having data transfer to and from the real world, specialized high speed arithmetic, and multiple access memory architectures. DSPs have some kind of computing engine, a program memory, a data memory, and some kind of input and output that connects it to the outside world.

To understand DSP we investigated one of the original specialized DSP chips on the market released in 1983. We chose to recreate and simulate the TMS32010. We utilized the structure and implemented controls to run 17 of the instructions. We also made an assembler to create the machine code to test the veriloged DSP.

2 Why: Digital Signal Processing

Digital Signal Processing (DSP) is imperative to our current way of life. It is seen in audio signal processing, digital image processing, video compression, speech recognition and more. We chose to do digital signal processing because it was a new topic that would develop our understanding of computer architecture. We were excited to have the opportunity to implement it in verilog. Some cool applications of DSP are speech coding and transmission in digital mobile phones, weather forecasting, economic forecasting, seismic data processing, and medical imaging such as CAT scans and MRI.

3 What is: Digital Signal Processing

Digital Signal Processing is characterized by having data transfer to and from the real world, specialized high speed arithmetic, and multiple access memory architectures. DSPs have parallel multiply and add capabilities, multiple memory accesses (to fetch two operands and store the result), lots of registers to hold data temporarily, efficient address generation for array handling, and special features such as delays or circular addressing.

What makes a DSP special from other CPUs is that it is able to receive and transmit data in real time, without interrupting its internal mathematical operations. DSPs are also getting three sources of data from the real world! These are signals coming in and going out, communication with an overall system controller of a different type, and communication with other DSP processors of the same type.

The most common components of a DSP are an instruction memory, data memory, ALU, registers, shifter, tristate buffer, control unit, and MAC.

Many digital signal processors employ a modified Harvard architecture for data memory. This model has the instruction memory and the data memory in physically separate memory storage. Because they use different data paths you can fetch an instruction from memory while simultaneously reading or writing to data memory.

This contrasts the von Neumann architecture that we became familiar with in class where the instructions and and data are stored in the same memory. The Harvard architecture is a faster model because instruction fetch and data memory operations can run in parallel.

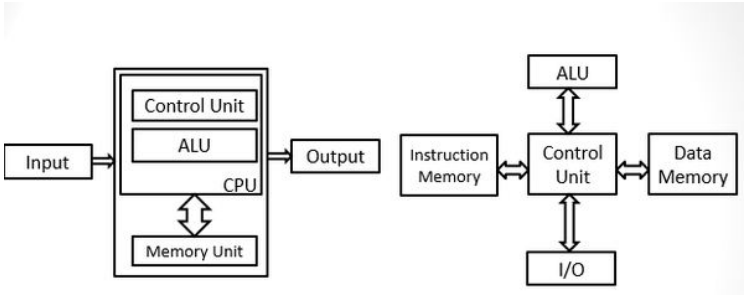

Figure 1: The von Neumann architecture is on the left and the Harvard architecture is on the right.

4 What We Did and How We Did It

We began by researching the basics of digital signal processing. Because two out of three of the members of the team were not familiar with digital processing, this research included much of the mathematical basics of the field as well as the computer architecture implementations. By analyzing and recreating the TMS32010, an early Texas instruments Digital Signal Processor, we were able to understand both the datapath of the Harvard Architecture as well as the hardware representation of the mathematical processes in digital signal processing. We aimed to understand the datasheet in full, recreate the chip in verilog, understand the datapath so that we could write a test bench that would showcase our understanding of the chip architecture, datapath, and our ability to simulate it.

The TMS32010 is a historically important chip that came out on April 8, 1983. It was not the first DSP chip, but it was the fastest for its time. It would compute a multiply operation in 200 nanoseconds. It was also able to execute instructions from both on-chip ROM(read only memory) and off-chip RAM(random-access memory). The other chips at the time had only canned DSP functions.

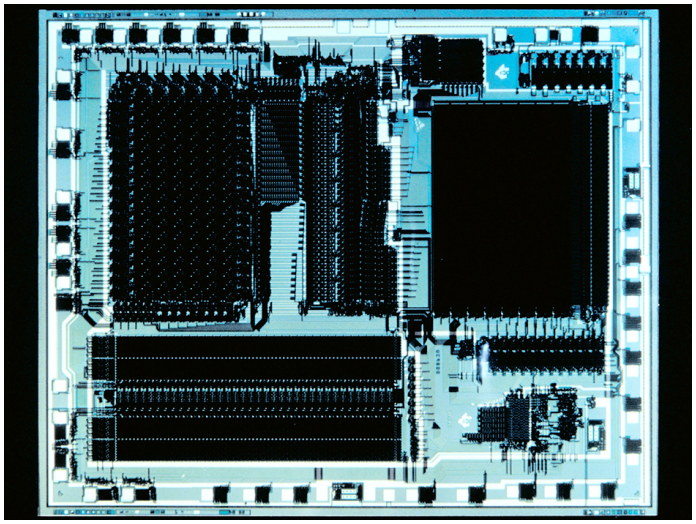

Figure 2: Photo of TMS32010

We utilized the data sheet for this chip to make it in verilog. There were many implicit muxes and controls that were not shown on the schematic in the datasheet. In order to determine where these unseen components were and how to operate them

we had to understand the datapath of the DSP and generate the controls ourselves. This was done by selecting several instructions from the instruction set and trying to draw the data paths by hand. This was a very effective way of understanding what was going on in the schematic and gave us a good intuition on how to build on it.

4.1 Schematics

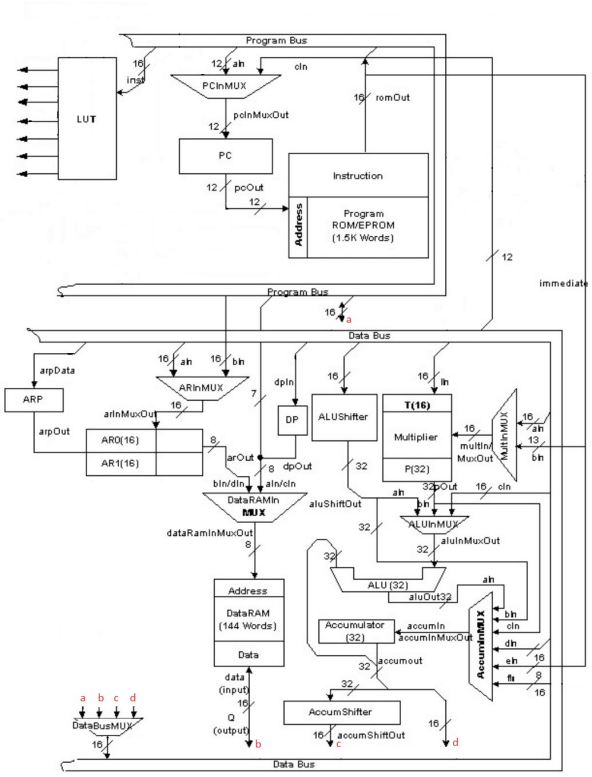

Figure 3: This is the TMS32010 schematic. This is the schematic that we wrote our assembler and verilog for. There is a larger version of this in the appendix.

4.2 Verilog

In modeling the TMS32010 architecture, data path, and instructions, we referenced the official TMS32010 User's Guide to stay as close as possible to the function and form of the original chip, although some liberties were taken in interpreting the somewhat convoluted and inconsistent documentation. We chose 16 instructions

(See Section 4.2.11 Assembler) to map the data path of; as a result, some modules, like the parallel shifter, are modeled but not used, while others, like the external ports, are not modeled at all. Our code is modular to easily incorporate more instruction sets. The code for our project can be found at `https://github.com/vivienyuwenchen/DSP`

Since a digital signal processor is a subset of CPU, we were able to reuse modules written in previous labs. These included the arithmetic logic unit (ALU), multiplexers (muxes), D flip-flops (DFFs), and memory units. Specific to DSPs are the multiplier, accumulator, barrel and parallel shifters, as well as a control look up table specific to our version of TMS32010. Each submodule, as well as the top level module, is explained in the following sections.

4.2.1 Multiplier (and T Register and P Register)

The multiplier takes in two 16-bit inputs and outputs a 32-bit product. It is connected to two registers, the T register and the P register. The T register is loaded with a 16-bit multiplicand from data memory through a separate instruction, i.e. Load T Register (LT) or related instruction, while the P register stores the 32-bit product. The multiplier is either a 16-bit number from data memory, in the case of a Multiply (MPY) instruction, or a sign-extended 13-bit immediate constant from the Multiply Immediate (MPYK) instruction.

Modern DSP designs combine the multiplier and the accumulator (explained in the next section) into a multiplier-accumulator or MAC unit. This speeds up and consolidates instructions, instead of requiring, for example, both a Load T and Accumulate Previous Product (LTA) instruction and an MPY instruction to execute a multiply-accumulate function, which is very common in DSP.

4.2.2 Accumulator

When enabled, the accumulator accumulates or adds the 32-bit input to the 32-bit stored value at every positive clock edge. The first 16 bits, 31 through 16, is the high-order word, and bits 15 through 0 is the low-order word. Depending on the instruction, either the high-order word or the low-order word can be stored in data memory.

The accumulator also has three controls: reset, load, and abs or absolute value. The reset control is typical of accumulators; it sets all bits to zero. The load control loads the input, replacing the current value instead of adding to it. This function

could also be executed by resetting the accumulator before allowing the normal accumulate function, but with our current setup, this would take two clock cycles to complete so we decided to make it a separate control. The absolute value function could also be done through other means, such as through the ALU. Again, our setup was not conducive to these means, as we did not include overflow, negative, etc. flags in the accumulator, which would normally be stored in a status register. Thus, we modeled this function behaviorally within the module, controlled by the abs control wire.

4.2.3 Barrel Shifter

The barrel shifter sign extends and left shifts the 16-bit input 0 to 15 places, zero-filling the lower order bits. The number of places shifted is determined by a 4-bit input from the instruction. The result is an arithmetically left-shifted 32-bit output.

We modeled this function in behavioral Verilog in two lines. In hardware, a barrel shifter for an n-bit word would be implemented by a cascade of $n \log_2 n$ parallel 2-to-1 multiplexers. For our 16-bit words, this would mean 64 multiplexers. The cascade of multiplexers allows shifting to be done in one clock cycle regardless of the number of places shifted.

4.2.4 Parallel Shifter

The parallel shifter left shifts the 32-bit input either 0, 1, or 4 places and outputs the first 16 bits of the result. It is used only in the Store High-Order Accumulator Word (SACH) instruction, which our model does not currently support. The 1- and 4-bit shifts are used in conjunction with multiply instructions.

4.2.5 Instruction Memory (and Program Counter)

The instruction memory supports up to 4096 16-bit words of external memory, indexed as 8-bit bytes by convention. As previously stated, we did not model external ports, so our instruction file is loaded through the test bench.

Unlike in our single-cycle and pipeline CPUs, the instruction memory of DSP is separate from the data memory in accordance to modified Harvard architecture. This structure increases speed, as it allows the device to fetch the instruction and the data at the same time.

In addition, in our previously modeled CPU designs, we had our program counter count up by 4's, as we had 32-bit words indexed by 8-bit bytes. With 16-bit words in our DSP, our program counter now counts up by 2's.

4.2.6 Data Memory

The data memory consists of 144 16-bit words. As we did not model any instructions that store data memory, we pre loaded our data memory with data before running the instructions our DSP does support.

4.2.7 Buses for Instruction and Data

We modeled the instruction and data buses as wires, with its inputs controlled by a mux. This was a design decision we made given the amount of time we had. Another method to implement this would have been to use a tri-state buffer for each of the inputs to the data bus.

4.2.8 Miscellaneous Registers Not Previously Mentioned

Two 16-bit auxiliary registers (AR) serve as temporary storage, indirect addressing of data memory, and loop control. The appropriate register is selected with the Auxiliary Register Pointer (ARP), the value of which is obtained from the instruction. Normally, the ARP is stored in the status register, which also contains the Data Memory Page Pointer (DP), as well as the Accumulator Overflow Flag Register (OV), Overflow Mode Bit (OVM), and Interrupt Mask Bit (INTM). However, since we did not model the latter three, we store ARP in its own 1-bit register.

Similarly, the DP is stored in its own 1-bit register. It is the first bit of direct addressing data memory.

4.2.9 ALU

We used the ALU we wrote in a previous lab, which served us previously in our single-cycle and pipeline CPU. It executes addition (ADD), subtraction (SUB), exclusive OR (XOR), set less than (SLT), AND, NAND, NOR, and OR. The first operand is the accumulator output, while the second one is determined by a mux depending on the address.

4.2.10 Control Look Up Table (and Muxes)

A variety of muxes exist to determine inputs for modules such as the multiplier, the accumulator, the ALU, the AR, and the PC. The control for these wires for each of our supported instructions are listed below. There are also control wires to enable writing to certain registers, as well as control the function of the ALU and the accumulator.

	ABS	APAC	PAC	SPAC	ZAC	ADDH	ADDS	AND	LACK
tReg_ctrl	0	0	0	0	0	0	0	0	0
pReg_ctrl	0	0	0	0	0	0	0	0	0
accumReset_ctrl	0	0	1	0	1	0	0	0	0
multInMux_ctrl	0	0	0	0	0	0	0	0	0
aluInMux_ctrl	00	00	00	01	00	00	00	01	00
accumInMux_ctrl	011	010	010	000	000	011	011	000	100
arInMux_ctrl	0	0	0	0	0	1	1	1	1
dataMux_ctrl	0	0	0	0	0	0	0	0	0
dataRamIn_ctrl	0	0	0	0	0	0	0	1	1
pcInMux_ctrl	11	11	11	11	11	11	11	11	11
alu_ctrl	000	000	000	001	000	000	000	100	000
load_acc	0	0	1	0	0	0	0	0	0
abs_acc	1	0	0	0	0	0	0	0	0
enable_acc	1	1	0	1	1	1	1	1	1
databus_ctrl	00	00	00	00	00	01	01	01	00
dataWr_ctrl	0	0	0	0	0	0	0	0	0
dp_ctrl	0	0	0	0	0	0	0	0	0

	OR	LDP	LT	LTA	MPY	ADD	LAC	SUB
tReg_ctrl	0	1	1	0	0	0	0	0
pReg_ctrl	0	0	0	1	1	0	0	0
accumReset_ctrl	0	0	0	0	0	0	0	0
multInMux_ctrl	0	0	0	0	0	0	0	0
aluInMux_ctrl	11	00	00	01	00	11	00	00
accumInMux_ctrl	000	011	000	000	000	001	010	000
arInMux_ctrl	1	1	1	1	1	0	0	0
dataMux_ctrl	0	0	0	0	0	0	0	0
dataRamIn_ctrl	1	1	1	1	1	1	0	1
pcInMux_ctrl	11	11	11	11	11	11	11	11
alu_ctrl	000	000	000	000	000	000	000	001
load_acc	0	0	0	0	0	0	1	1
abs_acc	0	0	0	0	0	0	0	0
enable_acc	0	0	0	1	0	1	1	1
databus_ctrl	01	01	01	01	01	01	01	01
dataWr_ctrl	0	0	0	0	0	0	0	0
dp_ctrl	0	1	0	0	0	0	0	0

4.2.11 Understanding the Instruction Set and Assembling

Computer Architecture has advanced a great deal since the original design and release of this chip. Current instruction sets like MIPS and ARM are not compatible with this schematic because the architectures were not yet founded when this chip was created therefore the chip was not designed with these instructions in mind. The instruction set used for this chip involves mostly single cycle single word instructions and are all 16 bits long. According to the data sheet this instruction set permits execution rates of five million per second.

Instructions that we implemented In order to write the machine code needed to run instructions on this chip we wrote an assembler that parsed the assembly instructions specifically made for this chip. To structure the assembler we broke the instructions into 4 different kinds. This is similar to the MIPS R-type, I-type, J-type convention, however the types we created were not stated in the data sheet. We broke up these instructions into shift types, data address types, immediate types, and all types.

We focused on the accumulator instructions, the auxiliary register and data page pointer instructions, and the T Register, P register, Multiply Instructions.

- "Shift" types: Shift type instructions are 16 bits long. Bits 15-12 include the OPP code of the instruction. Bits 11-8 contain how much you want to shift by. Bit 7 is a pointer for whether you are accessing direct register or an auxiliary register. Bits 6-0 are the data memory address. Shift type instructions access the contents of the data memory and then shift them as part of the instructions operation.

- "Data Address" Types: Data Address type instructions are 16 bits long. Bits 15-8 include the OPP code of the instruction. Bit 7 is a pointer for whether you are accessing direct register or an auxiliary register. Bits 6-0 are the data memory address. This kind of instruction is used for any operation that accesses the content of data memory but does not require a shift. Some examples of data address type instructions are ADDS, LDP, and LT.

- "Immediate" Types: Immediate types are those instructions that utilize constants. They are 16 bit instructions where bits 15-8 are OPP codes and 7- 0 are the 8 bit constant. This is often denoted as "k". LACK loads the accumulator with an immediate. This is an example of an Immediate type instruction.

- "All" Types: All type instructions are 16 bits long and all 16 bits are OPP code. There is no variable section that allows for selection of any kind like selection of an immediate or selection of a data memory address. Some examples of all type instructions are ABS, APAC, PAC, and SPAC. These examples are independent of any kind of variable inputs.

The instructions this can run:

Type	Name	16 Bit Break Up	Description
All	ABS	['0111111110001000']	If (ACC) <0, Then -(ACC) → ACC
S	ADD	['0000', 's', '0', 'd']	(ACC)+(dma) $\times 2^{shift}$ → ACC
D	ADDH	['01100000','0','d']	(ACC)+(dma) $\times 2^{16}$ → ACC
D	ADDS	['01100001', '0','d']	(ACC)+(dma) → ACC
D	AND	['01100001', '0','d']	data is zero padded AND (ACC) → ACC
S	LAC	['0010','s','0', 'd']	(dma) $\times 2^{shift}$ → ACC
K	LACK	['01111110','k']	constant → ACC
D	OR	['01111010','0', 'd']	data is zero padded OR (ACC) → ACC
S	SUB	['0001', 's', '0','d']	(ACC)-[(dma)$\times 2^{shift}$] → ACC
D	LDP	['01101111','0', 'd']	LSB of (dma) → DP(DP=0 or 1)
D	LT	['01101010','0','d']	(dma) → T
D	LTA	['01101100', '0', 'd']	(dma) → T, (ACC)+(P) → ACC
All	APAC	['0111111110001111']	(ACC)+(P) → ACC
All	PAC	['0111111110001110']	(P) → ACC
All	SPAC	['0111111110010000']	(ACC)-(P) → ACC
D	MPY	['01101101', '0', 'd']	(T) × (dma) → P

4.3 Datapath examples

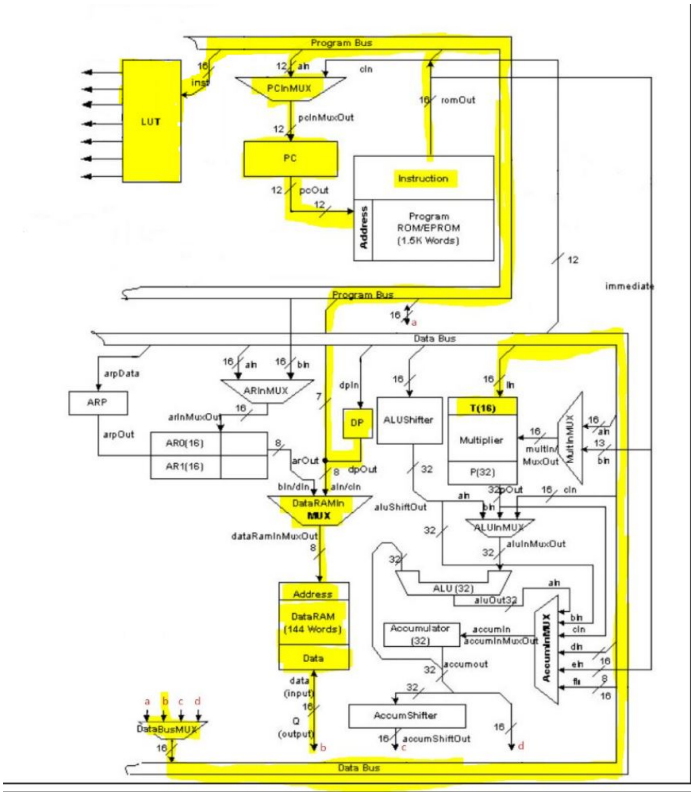

Figure 4: This is the datapath of the LT, Load T register, instruction

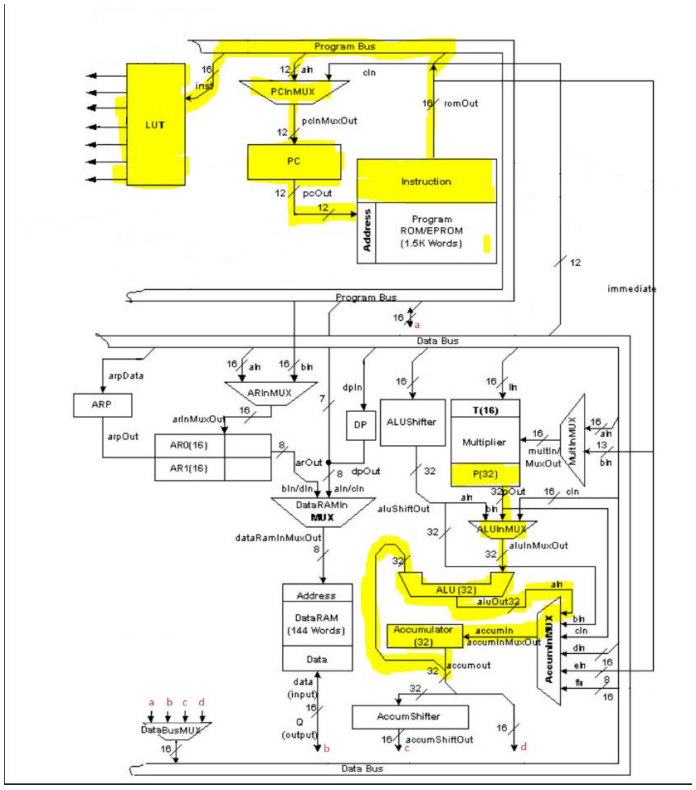

Figure 5: This is the datapath of the APAC, adding the p register with the value in the accumulator and storing it in the accumulator

Instructions that we did not implement The instructions we did not implement include the control instructions, the I/O instructions and the branch instructions.

There were several kinds of instructions we did not implement in our assembler however they have very interesting applications in our digital signal processing chip.

The TBLR(table read) and TBLW(table write) instructions were not included in our instruction set but they would allow crossover between the data memory and the program memory. TBLR(table read) transfers words from program memory to RAM. TBLW tranfers data from an internal RAM to an external RAM. These instructions are important in DSP because it permits the accesses of a large amount of RAM which is essential for pattern recognition like speech and image processing.

This chip also has branching capabilities and a set of instructions that support branching. We did not focus on these instructions because we have worked with branching in both of our previous CPU projects.

4.3.1 The Assembler Code

The assembler was written in python to parse assembly instructions and convert them into machine code using the instruction set specific to this chip. Splitting the instructions into the 4 discrete groups above made for more efficient decoding.

Each instruction was first translated from assembly code to arguments. These arguments included op codes, immediate, data memory addresses, and shifts. Based on the number of arguments found, it was classified by type. Instructions in English get converted to opcode in binary using a look up table. Each instruction of like types were translated in the same way, converting integer arguments to bit strings.

We based our assembler off of a Python MIPS compiler found at `https://github.com/zeckendorf/Python-MIPS-Assembler.git`.

4.3.2 Testing

Our test cases includes an assembly files that contains all of the intended instructions.

Using the assembler you can write an assembly file, run the command $python assembler.py ¡path-to-file¿.asm This creates a file called machine_code.c.dat. This file includes 16 bit binary instructions for the Digital Signal Processor. Below is an example of a test with an "all" type instruction, a "Data address" type instruction, and an "immediate" type instruction. This small test also uses both the data memory, the alu, the accumulator, and the Multiplier, all the crucial components of the digital signal processor. This small subset is included as the test dsp_test_quick.asm. The explanations of each instruction are to the right of the wave form and the translated machine code is included above the instruction bits.

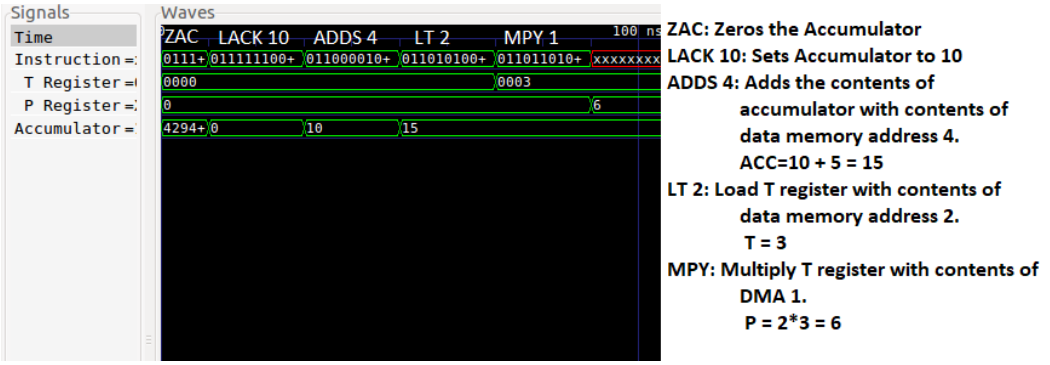

Figure 6: Above is a simple test of each of the different types of instruction. The assembly instruction is in white and the explanations are in black text.

A larger test is also included in the github. It is called dsp_test_full.asm. This test includes all of the instructions and types.

Tests were checked for correctness through GTK wave.

5 Reflection

In this section we will reflect on the workplan, the challenges we ran into, and the project as a whole.

5.1 Work Plan

	Expected Time	Actual Time
Research: Understand DSP Basics	3 hours	5 hours
Research: Architecture	3 hours	5 hours
Understand Filters	2 hours	1 hour
Design Data Path	3 hours	0 hours
Design Module Schematics	6 hours	0 hours
Midpoint Check-in	0.5 hours	0.5 hours
Update Schematics	2 hours	0 hours
Write Assembler and Test	0 hours	11 hours
Verilog Modules and Test	10 hours	15 hours
Refine Slide Show Presentation	3 hours	3 hours
Report and Write up	3 hours	8 hours
Total	35.5 hours	48.5 hours

5.2 Difficulties and Challenges

We decided to do a project on Digital Signal Processing because we thought it would be an interesting application and jumping off point from our current CPU knowledge. That being said the majority of our team had little to no experience in signal processing prior to this project therefore much of the time at the beginning was spent researching to understand the field as a whole before we could even dive into the hardware representations of the math. Once we had a grasp on the field and its applications we felt more comfortable learning about the individual modules and hardware components that make digital signal processors unique.

This preliminary research took longer than expected due to this unfamiliarity with the topic. This led us to investigating an already manufactured DSP chip instead of trying to design our own architecture.

Through investigating this specific chip we learned valuable skills that we hadn't anticipated learning in this project, like parsing technical documentation and understanding assembly and machine code enough to write our own assembler. The confidence we have gained in tackling schematics and datasheets that seem scary (the one for our 1983 DSP chip was over 400 pages long) and being able to break them down into usable, understandable chunks is a skill we will take with us throughout our education and careers.

In our original work plan we hadn't thought ahead to testing and hadn't realized that we would not be able to use the MIPS instruction set. The lack of MIPS compatibility was a blessing in disguise because it made us write our own assembler. Writing our own assembler gave us a newfound appreciation for assembly languages and made us have a thorough understanding of how machine codes work and how they are constructed.

Because we had not anticipated writing our own assembler, our time estimations were really thrown off. Although our timing for individual tasks are very far off, the total time is only 23.7% off.

We had an especially difficult time making the jump from theoretical math to hardware units and then to understanding the data path on a schematic that had implicit components. We tackled these hurdles by hand stepping through the schematic, identifying the datapath and adding the missing components where we saw fit.

5.3 Overall Project Reflection

We learned an incredible amount on this project. We got to learn about different processor architectures, the mathematics and hardware implementations of digital signal processing, applications of the field as well as grew our understanding of assembly and machine code by designing and making an assembler. We grew our confidence and ability to read technical documents and in turn practiced our own technical communication skills.

6 Ways to Continue

If the project were to be extended we would add input output functionalities to our DSP to make it more realistic. The current system is very buildable and modular so this would not be too complicated to do, time just did not allow for it to happen for this deadline. Additionally we would add more instructions like branching and the table reading and writing. These few instructions would increase the functionality of the chip simulation by a great deal and allow it to have very interesting applications. To really extend the depth, the DSP could be designed from scratch and/or optimized for specific tasks.

7 Appendix

7.1 Resources Used

https://archive.org/details/bitsavers_tiTMS320xx985_13292501
http://meseec.ce.rit.edu/eecc722-fall2003/722-10-8-2003.pdf
https://www.slideshare.net/Abhishekt11/dsp-datapath
https://en.wikipedia.org/wiki/Barrel_shifter
https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/verilog/ver-unsigned-multiply-accumulator.html
https://www.scribd.com/doc/24177561/Verilog-Code-for-Mac-Unit
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.567.1157&rep=rep1&type=pdf
http://www.ti.com/lit/ds/symlink/tms320c25.pdf
https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation
https://spectrum.ieee.org/tech-history/silicon-revolution/chip-hall-of-fame-texas-instruments-tms32010-digital-signal-processor

https://en.wikipedia.org/wiki/Digital_signal_processing
https://www.allaboutcircuits.com/technical-articles/an-introduction-to-digital-signal-processing/

7.2 Larger Image