Configurable FFT Accelerated Processor

ASCII Arts

Alec Keehbler

Aksel Torgerson

Max Johnson

Reid Brostoff

Final Report

University of Wisconsin-Madison

Prof. Joshua San Miguel

TA. Winor Chen

Fall 2021

Table of Contents

1. Introduction

1.1 Project Overview

ASCII Arts initially intended to create a Fourier Transform accelerated processor with FIR Filter capabilities. Our team was inspired by the signal processing class ECE 203 at UW Madison in which we learned these concepts. We decided that we would like to be able to break down signals into their component frequencies using the Fast Fourier Transform (FFT) process. Our chosen use case for FFT was filtering. We intended to input the accelerator an FIR filter and run each of its 1024 data point chunks through the filter to remove certain frequencies. Unfortunately, due to time constraints we were unable to implement the filtering capabilities.

We integrated a general purpose processor with the accelerator design described above. The goal was to have the processor also perform as a control for the accelerator. Some important functionality includes making the inverse FFT and FFT operations blocking meaning that while one was calculating, if another came through the processor would stall then send in the next one after the previous one was complete. We also made it so that the processor could perform general purpose instructions while the accelerator was working on signal processing as long as the general purpose instruction did not access FFT data. The processor would also deal with exceptions so that errors could be handled with any bad user assembly code.

Throughout the duration of this project, our team was able to enhance our ability to not just implement a project given a specification, but to brainstorm and rigorously create that specification as well. Some notable topics covered are: 1024 radix point DFT, CPU & ISA design, Cache & Memory design and utilization, assembler creation, and direct memory access I/O.

2. Architecture

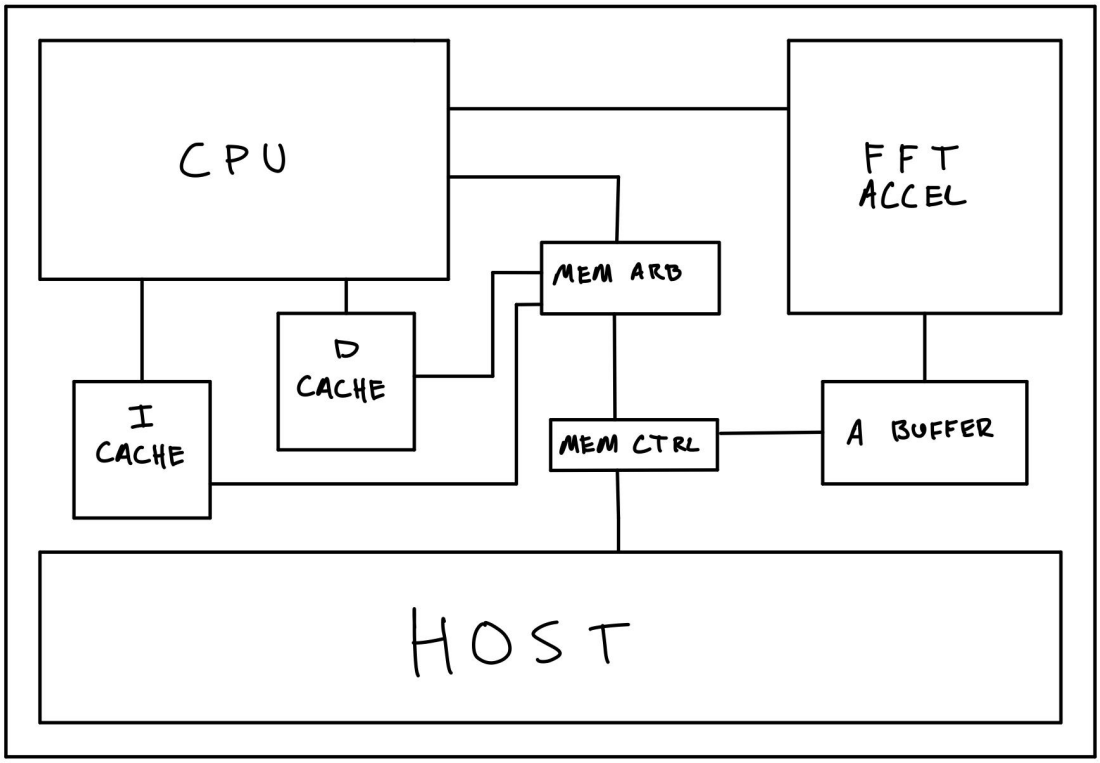


Figure 1: Top Level Architecture view.

2.1 CPU

The CPU used in this project is a custom single cycle design with an instruction set that was based off of the MIPS architecture. The design was made single cycle due to time constraints however it was implemented in a way such that it could be easily pipelined. There are five stages: fetch, decode, execute, memory, and writeback that are modularized in the design. The fetch stage retrieves the next instruction from memory. In decode, registers are read and control signals are set for the other stages. Execute contains the main ALU, does extension for immediates, and determines the next PC or address that the next instruction should be. The memory stage holds the data cache that is used as the CPU's memory. Finally, the writeback stage determines the data that will be written back to the given register destination. Overall, the CPU acts as a general purpose processor as well as a control for the FFT

accelerator unit. It also deals with exceptions, which currently halt the processor when an exception occurs.

2.1.1 Registers

Each register holds 32 bit values. The register file contains registers R0-R15 whereas the rest of the register are separate from the unit. Real data and imaginary data are laid out as fixed point data with 16 bits before the decimal point and 16 after. Only real registers can be used in memory addresses so that the user cannot put in imaginary numbers as addresses. This is handled by the assembler as it does not let registers higher than R7 to be used in those instructions.

Register Number	Register Name	Purpose
R0-R7	Real Number Registers	User registers used for real numbers
R8-R14	Imaginary Number Registers	User registers used for imaginary numbers
R15	Link Register	Link register used for JAL and JALR instructions
CR	Cause Register	Separate from the register file, this holds the reason for the exception*
EPC	Exception PC Register	Separate from the register file, this holds the address of the instruction that caused the exception
PC	PC Register	Separate from the reg file, holds the PC value

*More details on the encoding of the cause register in the exceptions section

2.1.2 ISA Summary

The ISA was designed for easily supporting general purpose software and was built around existing ISAs. It also includes special instructions to support the more specific purpose of performing an FFT with the STARTF, STARTI, and LOADF instructions. These instructions allow software to perform an

FFT with or without filtering as well as an inverse FFT with the specialized accelerator unit designed specifically for this task.

Opcode [31:27]	Instruction	Semantics
00000	HALT	Ceases Instruction issue, dump memory state to file
00001	NOP	None
00010	STARTF signalnumber, filter	Start FFT, if filter bit is set then this will be filtered
00011	STARTI signalnumber	Start iFFT (inverse FFT)
11111	LOADF signalnumber	Load the Filter with SigNum into the filter block
11000	ADD Rd, Rs, Rt	Rd <- Rs + Rt
11001	SUB Rd, Rs, Rt	Rd <- Rs - Rt
11010	XOR Rd, Rs, Rt	Rd <- Rd XOR Rt
11011	ANDN Rd, Rs, Rt	Rd <- Rs AND ~Rt
11100	SEQ Rd, Rs, Rt	if (Rs == Rt) then Rd <- 1 else Rd <- 0
11101	SLT Rd, Rs, Rt	if (Rs < Rt) then Rd <- 1 else Rd <- 0
11110	SLE Rd, Rs, Rt	if (Rs <= Rt) then Rd <- 1 else Rd <- 0
01000	ADDI Rd, Rs, immediate	Rd <- Rs + I(sign ext.)
01001	SUBI Rd, Rs, immediate	Rd <- I(sign ext.) - Rs
01010	XORI Rd, Rs, immediate	Rd <- Rs XOR I (zero ext.)
01011	ANDNI Rd, Rs immediate	Rd <- Rs AND ~I(zero ext.)
10000	ST Rd, Rs, immediate	Mem[Rs + I(sign ext.)] <- Rd

10001	LD Rd, Rs immediate	Rd <- Mem[Rs + I(sign ext)]
10011	STU Rd, Rs, immediate	Mem[Rs + I(sign ext.)] <- Rd Rs <- Rs + I(sign ext)
00100	J displacement	PC <- PC + 1 + D(sign ext.)
00101	JR Rs, immediate	PC <- Rs + I(sign ext.)
00110	JAL displacement	R15 <- PC + 1 PC <- PC + 1 + D(sign ext.)
00111	JALR Rs, immediate	R15 <- PC + 1 PC <- Rs + I(sign ext.)
01100	BEQZ Rs, immediate	if(Rs == 0) then PC <- PC + 1 + I(sign ext.)
01101	BNEZ Rs, immediate	if(Rs != 0) then PC <- PC + 1 + I(sign ext.)
01110	BLTZ Rs, immediate	if(Rs < 0) then PC <- PC + 1 + I(sign ext.)
01111	BGEZ Rs, immediate	if (Rs >=0) then PC <- PC + 1 + I(sign ext.)
10100	LBI Rs, immediate	Rs <- I(sign ext.)
10010	SLBI Rs, immediate	Rs <- (Rs << 16) | I(zero ext.)

2.1.3 Addressing Modes

2.1.3.1 Base + Offset

This mode uses the data stored in a register plus an immediate value as the effective address in memory or as the next instruction's address.

Ex. jr $1, 500

PC <- [Rs] + 500

Ex. st $5, $6, 500

Mem[R6 + 500] <- R5

2.1.3.2 Displacement

This addressing mode takes in an immediate value (displacement) plus pc plus one as the next instruction's address.

Ex. j 512

PC <- PC + 1 + 512

2.1.3.3 Immediate

Immediate values are signified by a decimal number (with no leading symbol). Depending on the instruction that is using the data, the value can be of varying lengths but will be sign extended or zero extended. The details for each immediates' length and extension can be found for each instruction below in the Instruction Descriptions section. As for an instruction addressing mode, the immediate will be added to pc plus one to get the new address of the next instruction

Ex. beqz $4, 200

First check if the R4 register's data is equal to 4. If it is then:

PC <- PC + 1 + 200

2.1.4 Instruction Formats

2.1.4.1 I Format Type 1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode					Rs				Rd				immediate																		

2.1.4.2 I Format Type 2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
opcode					Rs				immediate																						

2.1.4.3 R Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode					Rs				Rt				Rd				X (don't care)														

2.1.4.3 J Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
opcode					displacement																										

2.1.5 Instruction Descriptions

These are descriptions of each of the instructions present in our ISA. The syntax section for each instruction would be how to write the instruction in assembly using the assembler. There is asos an example for each instruction at the bottom of its section.

2.1.5.1 HALT

Syntax:

halt

Pseudo Code: (Can be in C/Verilog)
$stop()

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	x (don't care)																										

Usage and Examples:
The HALT instruction ceases instruction issue and dumps the memory state. It dumps the memory state by evicting all data cache values out to host memory that haven't been written back yet. All instructions before HALT execute normally however no instruction after will execute. The PC is left pointing to the instruction directly after the halt.
Ex. halt

2.1.5.2 NOP

Syntax:
nop

Pseudo Code: (Can be in C/Verilog)
no code- just do nothing

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	x (don't care)																										

Usage and Examples:
The NOP instruction occupies a position in the pipeline, but does nothing. Only practically used within the processor itself.
Ex. nop

2.1.5.3 STARTF - Start FFT

Syntax:
startf <signum>, <filter>

Pseudo Code: (Can be in C/Verilog)
start_accelerator_fft = true;
boolean filter = filter;
signal_to_calculate = signum;

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	signum																		filter	x (don't care)							

Usage and Examples:

This will start the FFT calculation on the signal number given and filter the FFT if the filter bit is set. If there are no signals being calculated it will be processed, otherwise the pipeline will be stalled. The pipeline will continue once the accelerator is done calculating and this startF will then be processed.
Ex. startf 1, 0x1; Start an FFT calculation on signal 1 and filter the output.

2.1.5.4 STARTI - Start Inverse FFT

Syntax:
starti <signum>

Pseudo Code: (Can be in C/Verilog)

start_accelerator_ifft = true;
signal_to_calculate = signum

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	signum																		0	x(don't care)							

Usage and Examples:
The STARTI instruction will do an inverse FFT calculation on the given signal number. If there is no signal being calculated, then this instruction will be processed and set the calculating signal to high. Otherwise, the instruction will stall the pipeline until the calculating signal is cleared.
Ex. starti 5; starts an inverse FFT calculation on signal number 5.

2.1.5.5 LOADF - Load Filter

Syntax:
loadf <signum>

Pseudo Code: (Can be in C/Verilog)
Mem <- signum_data

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	signum																		x(don't care)								

Usage and Examples:
The LOADF instruction loads the given signal number's data into the filter block on the accelerator so that data can be filtered. The input signal can only be 8Kb in size (1024 point sample). When this is used, it will set the filterLoaded signal high so that a STARTF instruction with the filtering bit high will no longer cause an exception.
Ex. loadf 10; loads the 10th signal into the filter block in the accelerator

2.1.5.6 ADD

Syntax:
add <Rd>, <Rs>, <Rt>

Pseudo Code: (Can be in C/Verilog)

Rd = Rs + Rt

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	Rs				Rt				Rd				x(don't care)														

Usage and Examples:

The ADD instruction will add the two operand register values (Rs and Rt) and store the result in Rd. If Rt, Rd, and Rs aren't all real or aren't all imaginary, then an exception will be raised.

Ex. add $2, $1, $3; Add together the real numbers in registers R1/R3 and store the result in register R2.

2.1.5.7 SUB

Syntax:

sub <Rd>, <Rs>, <Rt>

Pseudo Code: (Can be in C/Verilog)

Rd = Rt - Rs

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	Rs				Rt				Rd				x (don't care)														

Usage and Examples:

The SUB instruction will subtract Rt by Rs and store the result in Rd. If Rt, Rd, and Rs aren't all real or aren't all imaginary, then an exception will be raised.

Ex. sub R2, R1, R3; Subtract together the real numbers (R3 - R1) and store the result in register R2.

2.1.5.8 XOR

Syntax:

xor <Rd>, <Rs>, <Rt>

Pseudo Code: (Can be in C/Verilog)

Rd = Rs ^ Rt

Flags updated if executed:

None

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	Rs			0	Rt			0	Rd			x (don't care)														

Usage and Examples:

The XOR instruction will bitwise xor Rs and Rt together then store the result in Rd.
Ex. xor $2, $1, $3; Xor together the real numbers (R3 ^ R1) and store the result in register R2.

2.1.5.9 ANDN- And Not

Syntax:
andn <Rd>, <Rs>, <Rt>

Pseudo Code: (Can be in C/Verilog)
Rd = Rs && ~Rt

Flags updated if executed:
None

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	Rs			0	Rt			0	Rd			x (don't care)														

Usage and Examples:
The ANDN instruction will bitwise and Rs/~Rt (bitwise not) together then store the result in Rd.
Ex. andn $2, $1, $3; Bitwise not R3 then bitwise **and** R1/~R3 together and store the result in R2.

2.1.5.10 SEQ- Set if Equal

Syntax:
seq <Rd>, <Rs>, <Rt>

Pseudo Code: (Can be in C/Verilog)

```
if(Rs == Rt){
        Rd = 1;
} else{
        Rd = 0;
}
```

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	Rs			0	Rt			0	Rd			x (don't care)														

Usage and Examples:
The SEQ instruction checks to see if Rs and Rt are equal. If so it will set Rd to 1 , otherwise give Rd a value of 0.
Ex. seq $5, $6, $7; If the real numbers in R7 and R6 are equal then set R5 equal to 1, otherwise set it to 0.

2.1.5.11 SLT- Set if Less Than

Syntax:
slt <Rd>, <Rs>, <Rt>

Pseudo Code: (Can be in C/Verilog)

if(Rs < Rt){

Rd = 1;

} else{

Rd = 0;

}

Flags updated if executed:

None

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	Rs			0	Rt			0	Rd			x (don't care)														

Usage and Examples:

The SLT instruction checks to see if Rs is less than Rt. If so it will set Rd to 1 , otherwise give Rd a value of 0.

Ex. slt $5, $6, $7; If the real number in R6 is less than R7, set R5 to 1, otherwise set R5 equal to 0.

2.1.5.12 SLE- Set if Less Than or Equal

Syntax:

sle <Rd>, <Rs>, <Rt>

Pseudo Code: (Can be in C/Verilog)

if(Rs <= Rt){

Rd = 1;

} else{

Rd = 0;

}

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	Rs			0	Rt			0	Rd			x(don't care)														

Usage and Examples:

The SLE instruction checks to see if Rs is less than or equal to Rt. If so it will set Rd to 1, otherwise give Rd a value of 0.

Ex. sle $5, $6, $7; If the real number in R6 is less than or equal to R7, set R5 to 1, otherwise set R5 equal to 0.

2.1.5.13 ADDI- Add Immediate

Syntax:

addi <Rd>, <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

Rd = Rs + immediate;

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	Rs				Rd				immediate																		

Usage and Examples:

The ADDI instruction adds together the value in Rs with the **sign extended** immediate. If Rs and Rd aren't both real or both imaginary registers, then this will throw an exception.

Ex. addi $3, $2, 58; Adds together R2 and 58, then stores the result in R3.

2.1.5.14 SUBI- Subtract Immediate

Syntax:

subi <Rd>, <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

Rd = immediate - Rs;

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Rs				Rd				immediate																		

Usage and Examples:

The SUBI instruction subtracts the **sign extended** immediate by the value in Rs. If Rs and Rd aren't both imaginary or both real, then this will cause an exception.

Ex. subi $3, $5, 60; Subtract 60 by the value in R5 then store it in R3.

2.1.5.15 XORI- Xor Immediate

Syntax:

xori <Rd>, <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

Rd = Rs ^ Immediate

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	Rs			0	Rd			immediate																		

Usage and Examples:

The XORI instruction xor's the **zero extended** immediate by the value in Rs.

Ex. xori $3, $5, -35; XOR -35 and the value in R5 then store the result in R3.

2.1.5.16 ANDNI- And Not Immediate

Syntax:

andni <Rd>, <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

Rd = Rs && ~immediate

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	Rs			0	Rd			immediate																		

Usage and Examples:

The ANDNI instruction **and's** the value in Rs with the bitwise not value of the **zero extended** immediate then stores the result in Rd.

Ex. andni $1, $2, 80; Bitwise not the value 80 then and it with the value in R2 then stores the result in R1.

2.1.5.17 ST- Store

Syntax:

st <Rd>, <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

Mem[Rs + I(sign ext.)] <- Rd

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	Rs			Rd				immediate																		

Usage and Examples:

The ST instruction stores the value in Rd into the memory address at Rs plus the **sign extended** immediate.

Ex. st $3, $2, 100; Stores R3 into the memory address at R2 + 100.

2.1.5.18 LD- Load

Syntax:

ld <Rd>, <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

Rd <- Mem[Rs + I(sign ext.)]

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	Rs			Rd				immediate																		

Usage and Examples:

The LD instruction loads the value in memory at address Rs plus the **sign extended** immediate into Rd.

Ex. ld $3, $2, -50; Stores the memory value at R2 plus -50 into R3.

2.1.5.19 STU- Store and Update

Syntax:

stu <Rd>, <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

Mem[Rs + I(sign ext.)] <- Rd
Rs <- Rs + I(sign ext)

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	Rs			Rd				immediate																		

Usage and Examples:

The STU instruction stores the value in Rd to memory at the address Rs plus the **sign extended** immediate. It also saves the effective address (Rs plus immediate) into Rs after that has been done.

Ex. stu $1, $2, 60; Stores the value in R1 to the memory location at R2 plus 60. Then it saves the memory location (R2 +60) into R2.

2.1.5.20 J- Jump

Syntax:

j <displacement>

Pseudo Code: (Can be in C/Verilog)

goto

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	displacement																										

Usage and Examples:

The J instruction loads the PC with the value found by adding the PC of the next instruction (PC + 1) to the **sign extended** displacement.

Ex. j 562; Adds one to the PC and the displacement then stores that value in the PC register.

2.1.5.21 JR- Jump Relative

Syntax:

jr <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

goto (with more range)

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	Rs			immediate																						

Usage and Examples:
The JR instruction loads the PC with the value of Rs plus the **sign extended** immediate.
Ex. jr $3, 562; Adds the value in R3 with 562 into the PC register.

2.1.5.22 JAL- Jump and Link

Syntax:
jal <displacement>

Pseudo Code: (Can be in C/Verilog)
function();
next instruction;

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	displacement																										

Usage and Examples:
The JAL instruction first saves the address of the next instruction (PC + 1) into R15. It then stores the value PC plus 1 plus the **sign extended** displacement into the PC register.
Ex. jal 640; Saves PC + 4 into R8 then stores PC plus 1 plus 640 into the PC register.

2.1.5.23 JALR- Jump and Link Relative

Syntax:
jalr <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)
(Has more range than JAL)
function();
next instruction;

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	Rs			immediate																						

Usage and Examples:
The JALR first saves the address of the next instruction (PC + 1) into R15. Then the instruction stores the value of Rs plus the **sign extended** immediate into the PC register.
Ex. jalr $1, 20; Saves PC + 1 into R8 then stores R1 with 20 into the PC register.

2.1.5.24 BEQZ- Branch if Equal to Zero

Syntax:

beqz <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

if(Rs == 0){

PC = PC + 1 + immediate

}

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	Rs			immediate																						

Usage and Examples:

The BEQZ sees if the value in Rs is equal to zero, and if true then it stores the value of PC plus 1 plus the **sign extended** immediate.

Ex. beqz $1, 32; Sees if the value in R1 is equal to 0, if it is then it stores PC plus 1 plus 32 into the PC register.

2.1.5.25 BNEZ- Branch if Not Equal to Zero

Syntax:

bnez <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

if(Rs != 0){

PC = PC + 1 + immediate

}

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	Rs			immediate																						

Usage and Examples:

The BNEZ sees if the value in Rs is not equal to zero, and if true then it stores the value of PC plus 1 plus the **sign extended** immediate.

Ex. bnez $2, 64; Sees if the value in R2 is not equal to 0, if it is then it stores PC plus 1 plus 64 into the PC register.

2.1.5.26 BLTZ- Branch if Less Than Zero

Syntax:

bltz <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

if(Rs < 0){

PC = PC + 1 + immediate

}

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	Rs			immediate																						

Usage and Examples:

The BLTZ instruction sees if the value in Rs is less than zero, and if true then it stores the value of PC plus 1 plus the **sign extended** immediate.

Ex. bltz $0, 32; Sees if the value in R0 is less than 0, if it is then it stores PC plus 1 plus 32 into the PC register.

2.1.5.27 BGEZ- Branch if Greater Than or Equal to Zero

Syntax:

bgez <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

if(Rs>= 0){

PC = PC + 1 + immediate

}

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	Rs			immediate																						

Usage and Examples:

The BGEZ sees if the value in Rs is greater than or equal to zero, and if true then it stores the value of PC plus 1 plus the **sign extended** immediate.

Ex. bgez $1, 32; Sees if the value in R1 is greater than or equal to 0, if it is then it stores PC plus 1 plus 32 into the PC register.

2.1.5.28 LBI- Load Byte Immediate

Syntax:

lbi <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)

Rs = immediate

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rs				immediate																						

Usage and Examples:

The LBI instruction loads Rs with the **sign extended** immediate value.

Ex. lbi $12, 68; loads R12 with the value 68.

2.1.5.29 SLBI- Shift and Load Byte Immediate

Syntax:
slbi <Rs>, <immediate>

Pseudo Code: (Can be in C/Verilog)
Rs <- (Rs << 16) | I(zero ext.)

Encoding:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rs				x (don't care)							immediate															

Usage and Examples:
The SLBI instruction first shifts the value in Rs 16 bits to the left then replaces the lower 16 bits with the immediate value.
Ex. slbi $12, 4; shifts the value in R12 by 16 bits to the left, then replaces the bottom 16 bits with 0x04.

2.1.6 Exceptions

Exception handling is done within the CPU and with our current design, halts the processor. There is a cause register that holds a 32 bit value that describes which exception was raised. When there are no exceptions, the cause register will hold the value 32'h0. When an exception occurs, the PC of the next instruction will be saved into the EPC. In a full design, the specific exception would have a certain address associated with it that the PC would jump to and execute software instructions to handle it. However, due to time constraints the design just halts the processor from executing any more instructions. Below is a list of the exceptions that can be raised as well as their specific values associated with them in the cause register.

RealImagLoadException, 0x1:

Occurs when real register data is loaded into an imaginary register or vice versa.

ComplexArithmeticException, 0x2:

This exception will occur when an arithmetic operation (ADD, ADDI, SUB, SUBI) contains both

real and imaginary numbers.

FFTNotCompleteException, 0x4:

Occurs if access fft output memory addresses while data is being calculated on in the accelerator. FFT memory addresses were from 0x1000_0000 to 0x2FFF_FFFF.

MemoryAccessException, 0x8:

This exception will occur when illegal memory spaces are accessed. Data region addresses are from 0x3000_0000 to 0x4000_0000.

MemoryWriteException, 0x10:

This exception will occur when illegal memory spaces are written to. Data region addresses are from 0x3000_0000 to 0x4000_0000.

InvalidJMPException, 0x20:

This exception will occur when a JMP instruction jumps to an invalid PC. Instruction addresses can only be in the address range of 0x0 to 0x0FFF_FFFF.

InvalidFilterException, 0x40:

Occurs when there is a startF with the filter bit set high before a loadF has been done

2.2 Accelerator

The accelerator is a chip used to accelerate fast fourier transforms. It is a completely independent unit from the CPU, so the CPU and accelerator can run in parallel with each other. The only connection between the CPU and the accelerator is the STARTF instruction. The STARTF instruction is given to the CPU along with a signal number, which is then passed on to the accelerator via a single bus. The signal number indicates an 8kB signal stored in memory, which is grabbed by the memory controller and sent to the accelerator via a chunk buffer. The accelerator/signal data consists of 1024 real 32b values, each real value being paired with a 32b imaginary value. Once the accelerator has its data stored into its own memory, it runs through its process to completion and the output data is sent to the memory controller via another outward facing chunk buffer. While calculating, a signal is sent to the CPU to indicate that it is currently processing data. The CPU knows the accelerator is done computing once

that signal is set low, and other STARF instructions that are queued can be processed. The whole process, assuming a computation on a signal takes one cycle, takes 10 stages of 512 cycles each, plus 1024 cycles to load and 1024 cycles to store data to and from the FIFO's.

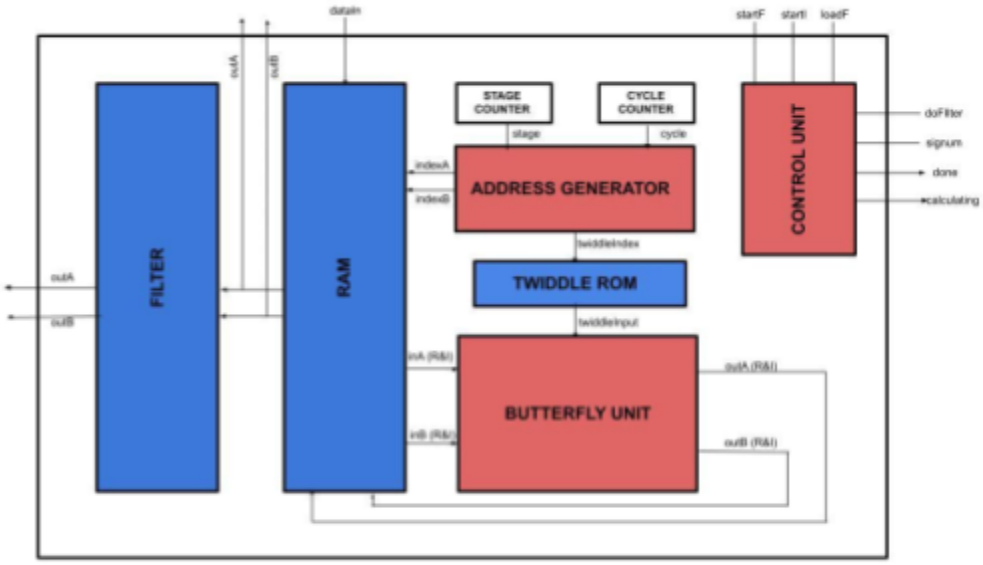


Figure 2: Accelerator Schematic

2.1.1 Butterfly Unit

The butterfly unit is the bread and butter of the FFT. It is the unit that is responsible for all calculations on the chip. It takes in 2 64b values (again each having a 32b real and a 32b imaginary part) along with a 64b twiddle factor value. The 2 values come in from the accelerator RAM, whose indices are chosen by the address generator. The twiddle factor that comes in is also determined by the address generator, which comes from the twiddle ROM. The output of the butterfly computation is then stored back into the RAM at the indices they came from, making the FFT implemented an "in-place" FFT.

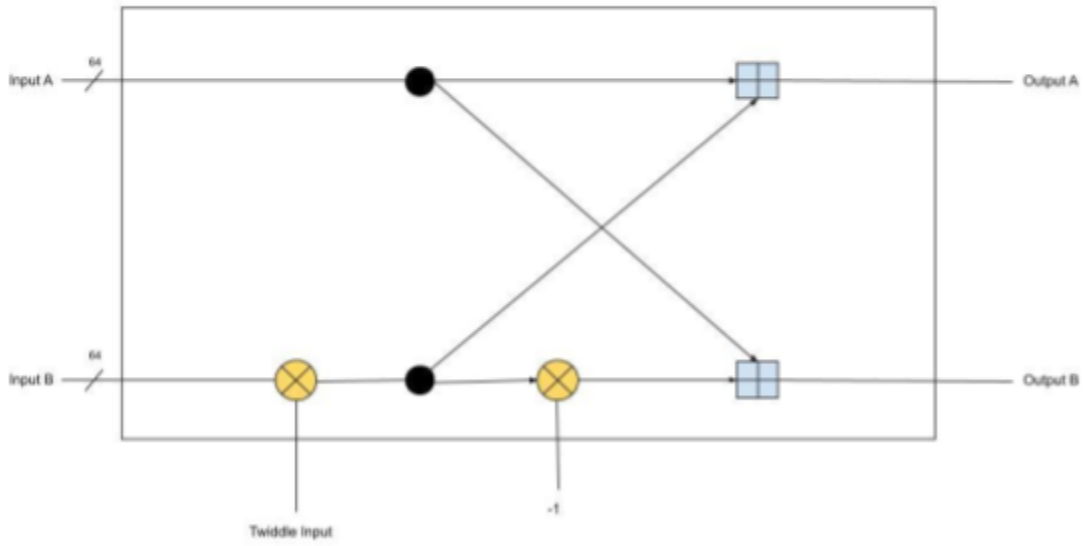


Figure 3: Butterfly Schematic

2.1.2 Address Generator

The address generator is responsible for choosing the indices necessary for the data being pulled from the RAM. It is also responsible for choosing which twiddle factor is needed for a computation in the butterfly unit. It chooses these indices based on the current stage and cycle of the current stage. In a 1024 Point Radix-2 DIT FFT, there are 10 stages of computations and each stage always has 512 computations. The algorithm can be thought of as each stage consisting of 2^{P-1} DFT algorithms, where the size of each DFT in each stage is determined as $1024/2^{P-1}$, where P is the stage. Using this knowledge, and knowing that each A input is matched with a B input that is $512/2^{P-1}$ indices higher than the A, the address generator can provide indices necessary to perform the butterfly computation each clock cycle. The equation used to determine the twiddle index for each cycle is as follows:

$$twiddleIndex = \left[\left[k * 2^{P-1}/1024\right]_{floor}\right]_{bit-rev} \quad where\ k \in 0, 1, 2..., N/2$$

Since these computations all follow base 2 operations, the address generator simply does bit shifts on the cycle count and stage count in order to generate its indices.

2.1.3 Dual-Port RAM

The dual-port RAM was responsible for holding the signal the FFT was being computed on. It was designed as a dual port due to the fact that the butterfly unit needed to read and store two signal values each cycle. The RAM also could be loaded by the in buffer by using an external index A, which would then load data through the RAM's A port. The same would go for loading data to the out buffer.

2.1.4 Control Unit

The control unit glued the whole FFT process together. It was responsible for arbitrating the signals coming from the CPU and the MC, and moving the accelerator through certain states of functionality. The control unit would hold the accelerator in an idle state until it received a signal to move into the loading state, then the calculating state, and finally the out load state. The whole FFT is a very linear process so there are not many complexities to the control unit.

2.1.5 Twiddle ROM

The twiddle ROM holds precomputed twiddle factors necessary for use in the butterfly unit. The twiddle factor given to the butterfly unit is determined by the index generated by the address generator.

2.1.6 Filter

The Filter is responsible for filtering the output FFT data, if necessary. If filtering is desired, the programmer should assert the LOADF signal. This feature was not implemented since problems were encountered in the creation of the algorithm.

2.3 Memory

Our memory structure is based upon a tri-cache design with the FPGA's host memory functioning as main memory. All cache requests are interfaced through an arbiter which prioritizes requests with a priority encoder. This priority encoder ensures that only one request is being serviced at a time, and that simultaneous requests are all serviced eventually. The Accelerator Buffer consists of an In Buffer and an Out Buffer. Our Address Space is pictured below; it is a 4GB region, that is 4B addressable. We chose to do this out of simplicity because the majority of the data we are operating on consists of 32b values. The Code and Static Data section is reserved for instructions and other immutable data. The FFT Data section is reserved for signal transform data, and can be read/written to by the accelerator or the CPU, but only when there is not a current transform happening. The Data Region is a multi-use space designed for the CPU to read/write data to. Our memory system is little endian and all accesses are word-aligned because every address space holds 4B. Currently in our design, on a HALT, the data cache does not write back all of its modified data. This was only left like this due to time constraints, but would be eventually implemented to dump all modified blocks upon a CPU HALT. The Accelerator Buffers, however, function independently from the CPU so they will still finish writing their contents to Host Memory upon a HALT.

Figure 4: Memory address space layout.

3. Micro Architecture

Below we have defined the interfaces for our micro architecture. The tables include the signal name, whether it is an input or output, the source or target (if it's an input where it came from and if it's an output, where the signal is going), and the semantics of each signal.

3.1 Integrated Processor

Below is the interface for the top level processor that includes modules for the CPU, Accelerator, and Memory Arbiter.

Signal Name	I/O	Source/Target Module	Semantics
clk	I	Global	Global clock signal
rst	I	Global	Global reset signal
[511:0] common_data_bus_in	I	Memory Controller	A block of data that was read from host memory
tx_done	I	Memory Controller	A signal to let the Arb know that a transaction has completed
rd_valid	I	Memory Controller	A signal to let the Arb know that the data on the bus is valid
[1:0] op	O	Memory Controller	An op code that lets the memory controller know if the request is a read, write, or none (idle)
[511:0] common_data_bus_out	O	Memory Controller	The data block that is being written to host
[31:0] io_addr	O	Memory Controller	The address of the data that is being read/written
[63:0] cv_value	O	Memory Controller	A control value code to

			let the memory controller know if the CPU halts or stalls

3.2 CPU

All CPU interfaces have their own subsection. This table is the top level CPU module.

Signal Name	I/O	Source/Target Module	Semantics
clk	I	Global	Global clock signal
rst	I	Global	Global reset signal
fftCalculating	I	Accelerator	Let's CPU know the accelerator is still calculating on a signal (blocks startF, or startI instructions until becomes low)
mcDataValid	I	Memory Arbiter	Lets the dCache know that the data from the memory arbiter is valid and ready to write
mcInstrValid	I	Memory Arbiter	Lets the iCache know that the data from the memory arbiter is valid and ready to write
[511:0] mcDataIn	I	Memory Arbiter	Data from the memory arbiter that will be written into the dCache
[511:0] mcInstrIn	I	Memory Arbiter	Data from the memory arbiter that will be written to the iCache
evictDone	I	Memory Arbiter	Lets the dCache know that eviction to the host memory has completed
halt	O	Memory Arbiter	Indicates a stop of execution and will dump the memory state
startI	O	Accelerator	Let's accelerator know that an inverse FFT needs to be performed
startF	O	Accelerator	Let's accelerator know that an FFT needs to be performed
loadF	O	Accelerator	Let's accelerator know that it needs to

			load in a filter
filter	O	Accelerator	Let’s accelerator know that it should filter the signal or not for a startF
[17:0] sigNum	O	Accelerator	Signal number for the accelerator
[511:0] dCacheOut	O	Memory Arbiter	The block to evict out of the dCache and written to host memory
dCacheEvict	O	Memory Arbiter	Let’s memory arbiter know that an evict needs to happen and write to host memory
[31:0] aluReuslt	O	Memory Arbiter	The result of the alu is the address into memory and is needed for the arbiter on data cache misses
exception	O	Memory Arbiter	Tells arbiter to dump the cache memory
cacheMissFetch	O	Memory Arbiter	Indicates miss in iCache and to retrieve instructions from host memory
cacheMissMemory	O	Memory Arbiter	Indicates miss in dCache and to retrieve data from host memory

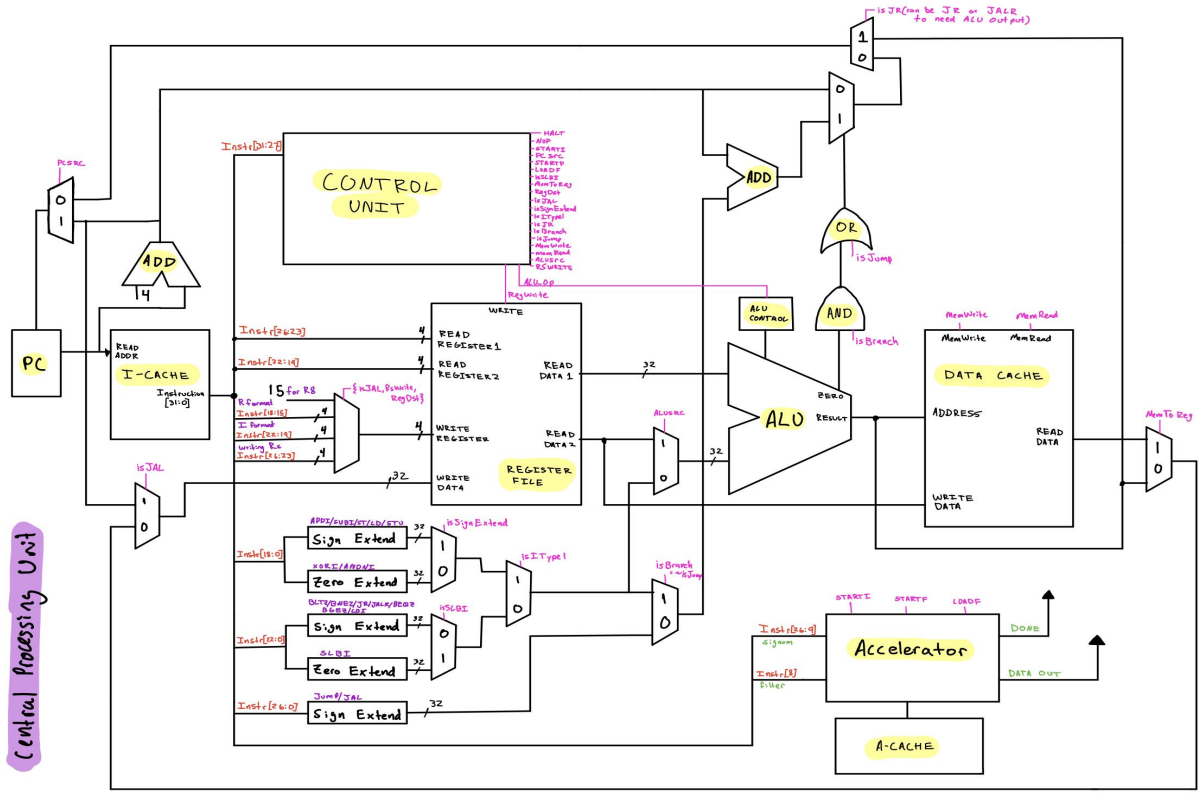


Figure 5: CPU Low level schematic.

3.2.1 Fetch Stage

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
halt	I	Decode Stage	Control signal that ceases instruction issue
[31:0] nextPC	I	Execute Stage	The next address that the pc should point to
stallDMAMem	I	Memory Stage	Stalls the PC if there is a DMA request being done
[511:0] mcDataIn	I	Memory Arbiter	Instructions coming into the cache if there is a miss

mcDataValid	I	Memory Arbiter	Control signal that signifies the mcDataIn is valid and can be put in the cache
blockInstruction	I	Decode Stage	Holds instruction constant if there is an issue with the instruction order
exception	I	Cause Register	Halts the processor if high
[31:0] instr	O	Decode Stage	The instruction to decode
[31:0] pcPlus1	O	Many	The current PC plus four (to get the next instruction if no jump or branch)
cacheMiss	O	Memory Arbiter	Lets the arbiter know that was an iCache miss and to start a request
[31:0] instrAddr	O	Memory Arbiter	Current address of the pc, needed for a DMA request

3.2.2 Decode Stage

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
[31:0] instr	I	Fetch Stage	The instruction to decode
[31:0] pcPlus1	I	Fetch Stage	The current PC plus one for branches and other instructions
stallDMAMem	I	Memory Stage	Stops writing registers on a memory DMA request
[31:0] writebackData	I	Writeback Stage	Data from writeback stage that needs to be written to registers
fftCalculating	I	Accelerator	Signal that lets stage know the accelerator is currently calculating

[31:0] read1Data	O	Execute Stage	Register data sent to execute stage
[31:0] read2Data	O	Many	Register data sent to execute stage
aluSrc	O	Execute Stage	Control Signal
isSignExtend	O	Execute Stage	Control Signal
isIType1	O	Execute Stage	Control Signal
isBranch	O	Execute Stage	Control Signal
halt	O	Many	Control Signal
nop	O	None	Signifies no operation for the processor on this cycle (useful for pipelined design)
memWrite	O	Memory Stage	Control Signal
memRead	O	Memory Stage	Control Signal
memToReg	O	Writeback Stage	Control Signal
isJR	O	Execute Stage	Control Signal
isSLBI	O	Execute Stage	Control Signal
isJump	O	Execute Stage	Control Signal
[3:0] aluOp	O	Execute Stage	Control Signal
startI	O	Accelerator	Control Signal
startF	O	Accelerator	Control Signal
loadF	O	Accelerator	Control Signal
blockInstruction	O	Fetch Stage	Control Signal
complexArithmeticEx	O	Cause Register	Exception if try to do arithmetic on a real and imaginary registers
realImageLoadEx	O	Cause Register	Exception if try to load real into imag reg or vice versa

invalidFilterEx	O	Cause Register	Exception if there is a startF with filtering bit high before a loadF has been done

3.2.2.1 Control Unit

Signal Name	I/O	Source/Target Module	Semantics
[4:0] opcode	I	Decode Stage	Determines instruction being done
fftCalculating	I	Accelerator	Control flag, that will set blockinstruction on a STARTF or STARTI
isJAL	O	Decode Stage	Control Signal
regDst	O	Decode Stage	Control Signal
rsWrite	O	Decode Stage	Control Signal
regWrite	O	Decode Stage	Control Signal
aluSrc	O	Execute Stage	Control Signal
isSignExtend	O	Execute Stage	Control Signal
isIType1	O	Execute Stage	Control Signal
isBranch	O	Execute Stage	Control Signal
halt	O	Many	Control Signal
nop	O	None	Control Signal (useful in pipelined implementation)
memWrite	O	Memory Stage	Control Signal
memRead	O	Memory Stage	Control Signal
memToReg	O	Writeback Stage	Control Signal
isJR	O	Execute Stage	Control Signal
isSLBI	O	Execute Stage	Control Signal
[3:0] aluOp	O	Execute Stage	Control Signal

isJump	O	Execute Stage	Control Signal
startI	O	Accelerator	Control Signal (if fftCalculating this gets a zero value even if startI)
startF	O	Accelerator	Control Signal (if fftCalculating this gets a zero value even if startF)
loadF	O	Accelerator	Control Signal
blockInstruction	O	Fetch Stage	Keep current instruction constant if there is an issue with the instruction order (set on a startF or startI and fftCalculating high)

3.2.2.2 Register File

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
[3:0] read1RegSel	I	Decode Stage	Determines which register to read from
[3:0] read2RegSel	I	Decode Stage	Determines which register to read from
[3:0] writeRegSel	I	Decode Stage	Signal to determine which register is going to be written to (if any)
[31:0] writeData	I	Decode Stage	The data being written to a register
write	I	Decode Stage	Control signal to determine if it is writing to a register or not
[31:0] read1Data	O	Execute Stage	Register data from first register being read
[31:0] read2Data	O	Many	Register data from second register being read

err	O	None	Error signal if register file failed (debug signal)

3.2.3 Execute Stage

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
[31:0] instr	I	Fetch Stage	Instruction used for immediate values
[31:0] pcPlus1	I	Fetch Stage	The current PC plus one used in calculations
[31:0] read1Data	I	Decode Stage	Register data from first register that was read
[31:0] read2Data	I	Decode Stage	Register data from the second register that was read
isSignExtend	I	Decode Stage	Control signal to determine if immediate values need to be sign extended
isIType1	I	Decode Stage	Control signal used to determine which immediate extended type
isBranch	I	Decode Stage	Control signal used to determine which offset to use for the PC
aluSrc	I	Decode Stage	Determines the B input to the ALU
isJump	I	Decode Stage	Determines if the instruction was a jump to put offset into the PC
isJR	I	Decode Stage	Determines if aluResult needed for nextPC or not
isSLBI	I	Decode Stage	Determines if the instruction will be zero extended or not

[3:0] aluOp	I	Decode Stage	Determines which ALU operation will be performed
[31:0] nextPC	O	Fetch Stage	The next PC to pass back to the fetch stage eventually
[31:0] aluResult	O	Many	The result of the alu which will be sent to the memory and writeback stage, as well as the memory arbiter
invalidJMPEx	O	Cause Register	Exception if jump to an invalid address

3.2.3.1 ALU

Signal Name	I/O	Source/Target Module	Semantics
[31:0] A	I	Execute Stage	The first input to the ALU
[31:0] B	I	Execute Stage	The second input to the ALU
[3:0] Op	I	Execute Stage	Determines which operation that the ALU will perform
[31:0] out	O	Many	The output data of the ALU
isTaken	O	Execute Stage	Determines if the branch was taken or not based off of the input to the ALU

3.2.4 Memory Stage

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
[31:0] aluResult	I	Execute Stage	Result of the alu that will address into memory

[31:0] read2Data	I	Decode Stage	Data from the second register that may be written to memory
memWrite	I	Decode Stage	Control signal on whether to write to memory
memRead	I	Decode Stage	Control signal on whether to read from memory
halt	I	Decode Stage	Control signal on whether to halt memory then dump its current state
exception	I	Cause Register	Control signal that an exception was raised and to dump the memory state
[511:0] mcDataIn	I	Memory Arbiter	Data coming into the cache if there is a miss
mcDataValid	I	Memory Arbiter	Control signal that signifies the mcDataIn is valid and can be put in the cache
fftCalculating	I	Accelerator	Control to show that the accelerator is calculating a signal
evictDone	I	Memory Arbiter	Signal from the memory arbiter that the eviction has completed
[31:0] memoryOut	O	Writeback Stage	Output data of reading from the data memory
cacheMiss	O	Memory Arbiter	Control signal for the MMU when there is a miss
[511:0] mcDataOut	O	Memory Arbiter	Data to be evicted out of the cache and written back to host memory
cacheEvict	O	Memory Arbiter	Control signal that lets arbiter know that mcDataOut is ready for eviction back to host memory
stallDMAMem	O	Many	Stalls the processor while the dma request is being done
memAccessEx	O	Cause Register	Exception if read outside the data region

memWriteEx	O	Cause Register	Exception if write outside the data region
fftNotCompleteEx	O	Cause Register	Exception if write or read fftData while the accelerator is calculating

3.2.5 Writeback Stage

Signal Name	I/O	Source/Target Module	Semantics
[31:0] memoryOut	I	Memory Stage	Output of reading from memory that may be needed back in the decode stage
[31:0] aluResult	I	Execute STage	Output of the alu that may be needed back in the decode stage
memToReg	I	Decode Stage	Control signal that determines whether the memoryOut data will be used back in the decode stage
[31:0] writebackData	O	Decode Stage	The data being sent back to the decode stage (that may be written to a register)

3.2.6 Cause Register

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
realImagLoadEx	I	Decode Stage	Exception if try to load real data into imaginary register or vice versa
complexArithmeticEx	I	Decode Stage	Exception if try to do arithmetic on a real and imaginary register

fftNotCompleteEx	I	Memory Stage	Exception if access fft data region while the accelerator is calculating
memAccessEx	I	Memory Stage	Exception if read an illegal memory address
memWriteEx	I	Memory Stage	Exception if write an illegal memory address
invalidJMPEx	I	Execute Stage	Exception if jump to an invalid address
invalidFilterEx	I	Decode Stage	Exception if do a startF instruction with filtering high before executing a loadF
[31:0] causeDataOut	O	None	Debug signal for exceptions (would be used in exception handling)
exception	O	Many	Control to show if an exception has been raised
err	O	None	Debug signal for if there was an error in the state

3.2.7 EPC Register

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
[31:0] epcIn	I	Fetch Stage	The address of the instruction after the one that caused an exception
write	O	Cause Register	Write EPC on an exception being raised
[31:0] epcOut	O	None	Useful for exception handling, would be the address to return to after the handling

3.3 FFT Accelerator

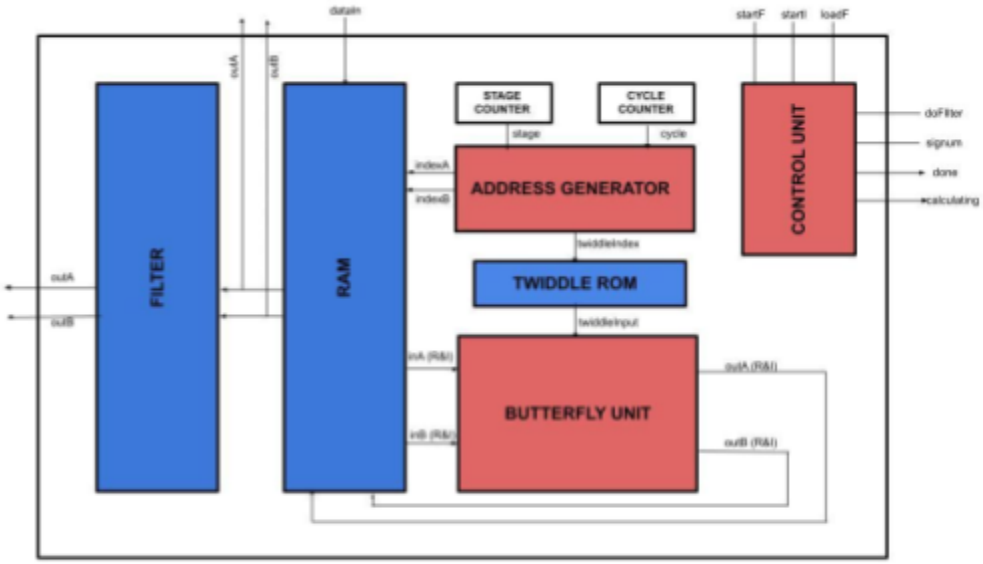


Figure 6: Accelerator Schematic

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
startF	I	CPU	Start FFT
startI	I	CPU	Start IFFT
loadF	I	CPU	Load Filter
filter	I	CPU	Start Filtering
loadInFifo	I	MC	Load input FIFO
transformComplete	I	MC	Data has been stored in host
[17:0] sigNum	I	CPU	Signal number of data
[511:0] mcDataIn	I	MC	Block of data being put into FIFO
accelWrBlkDone	I	MC	Block has been stored to host
done	O	CPU	FFT is done
calculating	O	CPU	FFT is calculating

[17:0] sigNumMC	O	MC	Signal number of data sent to MC
{511:0] mcDataOut	O	MC	Block of data sent to MC
outFifoReady	O	MC	Out fifo is full and can be read
mcOutDataValid	O	MC	Block of data sent is valid
inFifoEmpty	O	MC	In fifo is empty and ready to be filled

3.3.1 Butterfly Unit

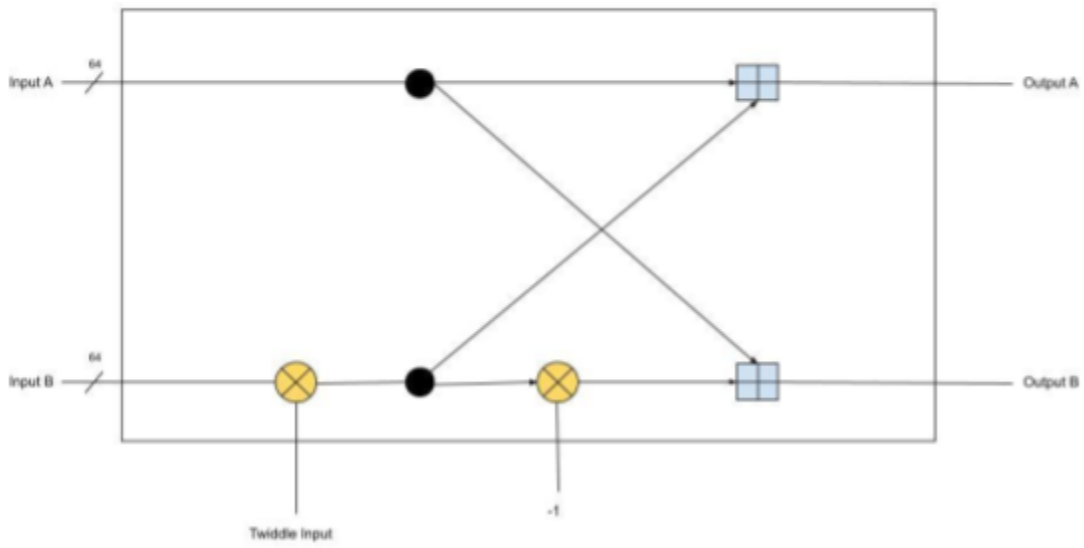


Figure 7: Butterfly Schematic

Signal Name	I/O	Source/Target Module	Semantics
[31:0] real_A	I	RAM	Real part of A
[31:0] imag_A	I	RAM	Imaginary part of A
[31:0] real_B	I	RAM	Real part of B
[31:0] imag_B	I	RAM	Imaginary part of B

[31:0] twiddle_real	I	Twiddle ROM	Real twiddle factor
[31:0] twiddle_imag	I	Twiddle ROM	Imaginary twiddle factor
[31:0] real_A_out	O	RAM	Real part of A
[31:0] imag_A_out	O	RAM	Imaginary part of A
[31:0] real_B_out	O	RAM	Real part of B
[31:0] imag_B_out	O	RAM	Imaginary part of B

3.3.2 Filter

Not Implemented, No I/O

3.3.3 Address Generator

Signal Name	I/O	Source/Target Module	Semantics
[4:0] stageCount	I	StageCounter	Current Stage
[8:0] cycleCount	I	CycleCounter	Current Cycle
[9:0] indexA	O	RAM	Index for A
[9:0] indexB	O	RAM	Index for B
[8:0] twiddleIndex	O	Twiddle ROM	Index for Twiddle ROM

3.3.4 Control Unit

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global Signal
rst	I	GLOBAL	Global Signal
startF	I	CPU	Start FFT
startI	I	CPU	Start IFFT

loadExternalDone	I	ACCEL	Done loading RAM from FIFO
doFilter	I	CPU	Filter
done	I	ACCEL	Calculation is done
outLoadDone	I	OUT FIFO	Loading to out FIFO is done
outFifoReady	I	OUT FIFO	Out fifo is ready to be emptied
startLoadingRam	I	IN FIFO	Start loading RAM from in FIFO
transformComplete	I	MC	Output has been stored
inFifoEmpty	I	IN FIFO	Input fifo is empty
[17:0] sigNum	I	ACCEL	Signal number of data
calculating	O	CPU	Accel is calculating data
loadExternal	O	RAM	wrEn for loading from FIFO
loadInternal	O	RAM	wrEn for loading from butterfly
writeFilter	O	FILTER	wrEn for filter
isIFFT	O	CPU/MC	Indicator for IFFT
fDone	O	ACCEL	Filter is done
aDone	O	ACCEL	Accel computation is done
loadOutBuffer	O	RAM	wrEn for out buffer

3.3.5 Twiddle ROM

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global Signal
rst	I	GLOBAL	Global Signal
[8:0] twiddleIndex	I	AGEN	Index to pull from ROM

[31:0] twiddleReal	O	Butterfly	Real part of the twiddle factor
[31:0] twiddleImag	O	Butterfly	Imaginary part of the twiddle factor

3.3.6 RAM

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global Signal
rst	I	GLOBAL	Global Signal
load	I	CONTROL	Load from butterfly
externalLoad	I	CONTROL	Load from FIFO
[31:0] real_A_i	I	BUTTERFLY	Real A
[31:0] imag_A_i	I	BUTTERFLY	Imaginary A
[31:0] real_B_i	I	BUTTERFLY	Real B
[31:0] imag_B_i	I	BUTTERFLY	Imaginary B
[31:0] real_A_o	O	BUTTERFLY	Real A out
[31:0] imag_A_o	O	BUTTERFLY	Imaginary A out
[31:0] real_B_o	O	BUTTERFLY	Real B out
[31:0] imag_B_o	O	BUTTERFLY	Imaginary B out

3.3.5 Stage Counter

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global Signal
rst	I	GLOBAL	Global Signal

[4:0] stageCount	O	AGEN	Count for the stage

3.3.5 Cycle Counter

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global Signal
rst	I	GLOBAL	Global Signal
[8:0] cycleCount	O	AGEN	Count for the cycle

3.4 Memory

3.4.1 Caching

3.4.1.1 Instruction Cache

The Instruction Cache is a read-only direct mapped cache. It is 8kB in size, structured with 128 64B lines. Its purpose is to hold CPU instructions and is directly mapped for simplicity because our use case does not contain many loops. Because instructions are 32b, each cache line can hold 16 instructions. There is no method for evicting lines because the data in the instruction section is immutable.

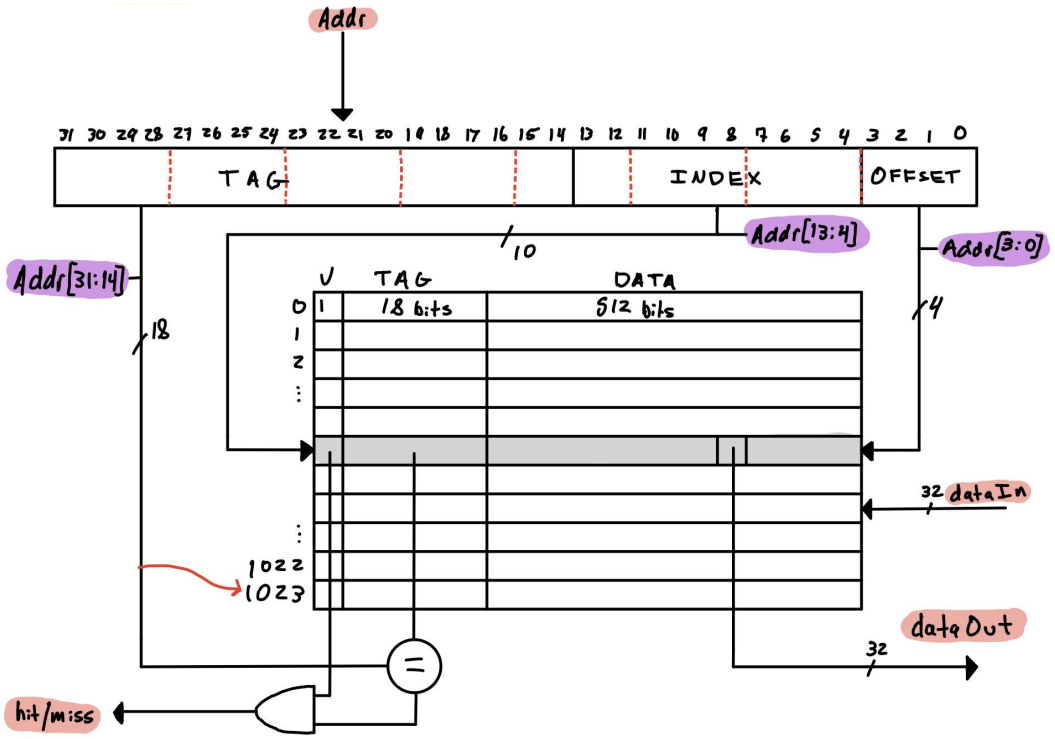


Figure 8: Instruction cache design schematic.

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
[31:0] addr	I	Fetch Stage	The address that the PC is pointing to
[511:0] blkIn	I	Mem Arb	Block of instruction data coming from host memory
ld	I	Fetch Stage	Signals to the cache after a miss that the incoming block is valid and ready to be loaded
[31:0] instrOut	O	Fetch Stage	The requested instruction for the CPU
hit	O	Fetch Stage	Lets the Fetch Stage know that the requested instruction has been found in the cache
miss	O	Mem Arb	Signals to the Mem Arb that we

			need the instruction that is located at addr. This request is queued and serviced by the Mem Arb.

3.4.1.2 Data Cache

The Data Cache is a read/write directly mapped cache. The implementation is similar to the Instruction Cache, but with added functionality to evict blocks that have been modified. It is 16kB in size and consists of 256 lines. Its purpose is to hold data that the CPU is reading from or writing to. If the CPU modifies data in the cache, the dirty bit is set to ensure that in the future if that line gets overwritten, the modified data is written back to host memory.

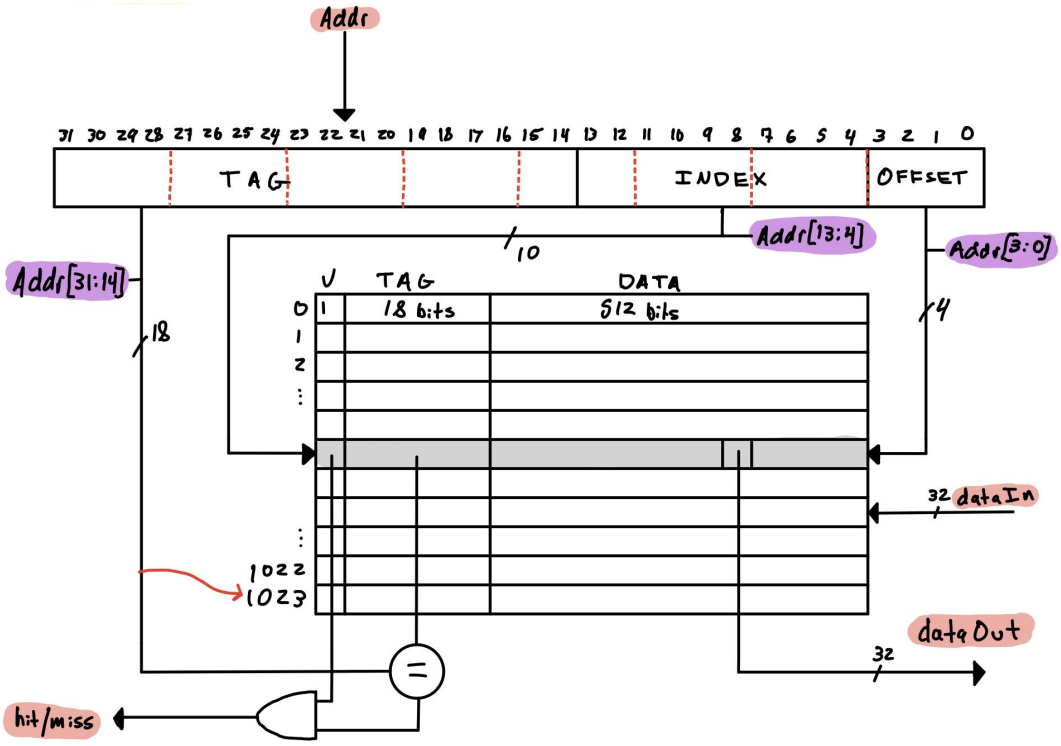


Figure 9: Data cache design schematic.

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
en	I	Memory Stage	Enables the cache upton a read or write
[31:0] addr	I	Fetch Stage	The data address in memory for a read or write request
[511:0] blkIn	I	Mem Arb	Block of instruction data coming from host memory
[31:0] dataIn	I	Memory Stage	The data word that is being written to memory
rd	I	Memory Stage	Signals to the cache that a rd access is requested
wr	I	Memory Stage	Signals to the cache that a wr access is requested
ld	I	Memory S	Signals to the cache after a miss that the incoming block is valid and ready to be loaded
[31:0] instrOut	O	Fetch Stage	The requested instruction for the CPU
hit	O	Fetch Stage	Lets the Fetch Stage know that the requested instruction has been found in the cache
miss	O	Mem Arb	Signals to the Mem Arb that we need the instruction that is located at addr. This request is queued and serviced by the Mem Arb.
evict	O	Mem Arb	Lets the Mem Arb know that we have a write request to host memory
[511:0] blkOut	O	Mem Arb	The data line that is being evicted back to host memory

3.4.1.3 Accelerator Buffer

The Accelerator Buffer is a two-part module composed of an In Buffer and an Out Buffer. The In Buffer is an 8kB buffer that holds an entire chunk of signal data. Each point in the signal is composed of a 32b real part, and 32b imaginary part. Each buffer line holds 1 data point and therefore is 64b. Since there are 1024 points in each signal chunk, the buffer is therefore 8Kb. The In Buffer takes in 512b blocks of data, and shifts them in as 8 64b values. Then, once the buffer is full, it will shift the data out one point at a time until it has shifted out all of its contents. The Out Buffer functions similarly, however when shifting data in, it takes each 64b point at a time shifting into the buffer, and then will shift out 512b every time the Mem Arb is ready for another request.

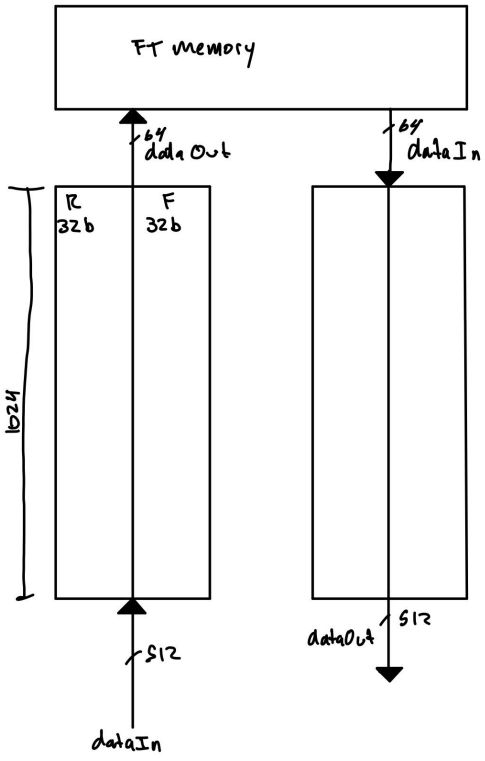


Figure 10: Accelerator Buffer top level diagram.

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
accelWrEn	I	Accelerator	Enables the buffer to shift in a data point from the Accelerator
mcWrEn	I	Mem Arb	Enables the buffer to shift in a block of data from host
[511:0] mcDataIn	I	Mem Arb	A block of signal data being written to the In Buffer
[63:0] accelDataIn	I	Accelerator	A transformed data point being written to the buffer from the accelerator
accelWrBlkDone	I	Mem Arb	Signals to out buffer that Host is ready for the next block
outBufferFull	O	Mem Arb	Signals to the Mem Arb that the buffer is full and is ready to start writing to host
inBufferFull	O	Accelerator	Signals to the accelerator that the buffer is full and is ready to shift out data points
[63:0] accelDataOut	O	Accelerator	The data point being shifted to the accelerator
[511:0] mcDataOut	O	Mem Arb	The data block being written to host memory
mcDataOutValid	O	Mem Arb	Signals to the Mem Arb that the data block is valid and can be written
accelDataOutValid	O	Accelerator	Signals the the accelerator that the data point is valid and can be shifted into the accelerator
inFifoEmpty	O	Accelerator	Tells the accelerator that the Fifo is empty

3.4.1 Memory Arbiter

The Memory Arbiter is responsible for arbitrating between read/write requests to/from host memory. Because an instruction request could come in simultaneously with an accelerator request or data request, the arbiter needs to prioritize these requests and ensure that all of them get serviced. Because there won't be a case where multiple requests from the same unit come in at the same time, the Arbiter uses a priority encoder which consists of 3 bits corresponding to the instruction cache, data cache, and accelerator buffer, respectively. The arbiter has a finite state machine inside that will respond to these requests in the following order: Accelerator read requests, accelerator write requests, data eviction (write) requests, data read requests, instruction read requests. The Arbiter also encapsulates a signal location lookup table. As signals get loaded into Host memory, the table keeps track of the starting block of the signal, and the ending block, then it will know where to locate the signal in memory given the sigNum, which is an integer representing the order in which signals were loaded in. Note, due to time constraints, our current design is set up to handle 1 8kB signal. The signal look up table is still in function, however, it always holds 0, as the starting block of the signal, and 0 as the ending block (because 1 8kB signal just takes up 1 of the 262,144 possible signal blocks.

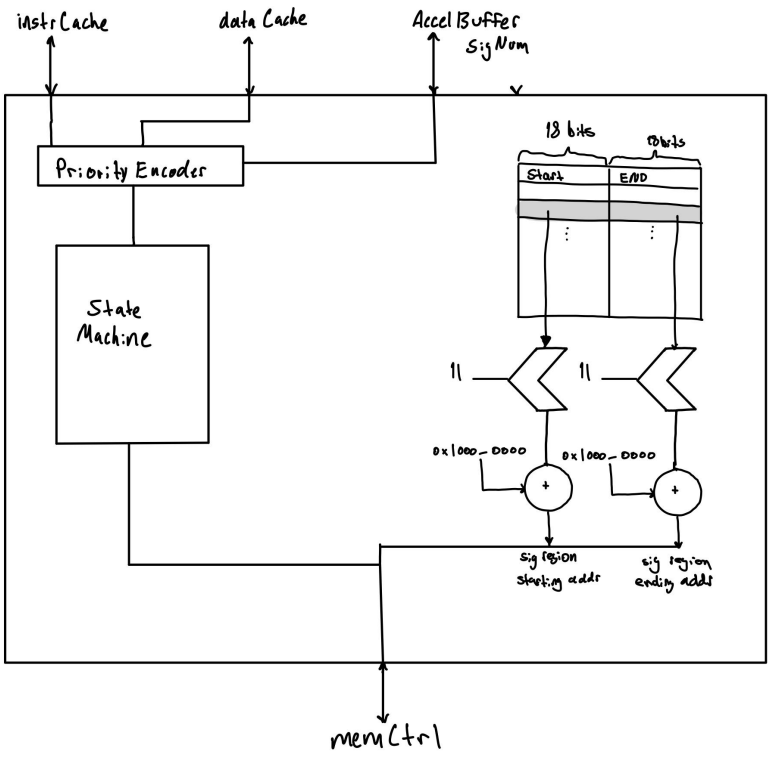


Figure 11: Memory arbiter schematic.

Signal Name	I/O	Source/Target Module	Semantics
clk	I	GLOBAL	Global clock signal
rst	I	GLOBAL	Global reset signal
dump	I	CPU	Signal from the CPU letting the Arb know to request all modified data blocks from data cache
instrCacheBlkRequest	I	Fetch Stage	A signal to let the Arb know an instruction block is requested
[31:0] instrAddr	I	Fetch Stage	The address of the requested instruction block
dataCacheBlkReq	I	Memory Stage	A signal to let the Arb know a data block is requested
[31:0] dataAddr	I	Memory Stage	The address of the requested data block
dataCacheEvictReq	I	Memory Stage	A signal to let the Arb know a

			data block write is requested
[511:0] dataBlk2Mem	I	Memory Stage	The block of data being evicted from the data cache
accelDataRd	I	Accelerator	Input to the accelerator to let it know that a block has been read successfully
accelDataWr	I	Accelerator	A signal to let the accelerator know that a block has been written successfully
[511:0] accelBlk2Mem	I	Accelerator	The block of data being shifted out from the accelerator to host memory
[17:0] sigNum	I	Accelerator	A signal number that corresponds to the signal data we want to operate on
[511:0] common_data_bus_in	I	Mem Controller	The data bus connecting the memory controller to host
tx_done	I	Mem Controller	A signal from the memory controller letting us know the transaction is done
rd_valid	I	Mem Controller	A signal from the memory controller letting us know the read is valid data
[511:0] instrBlk2Cache	O	Fetch Stage	The bus for instruction data that was requested
instrBlkCacheValid	O	Fetch Stage	A signal to let the fetch stage know the data on the bus is valid
dataEvictAck	O	Memory Stage	A signal to let the fetch stage know the evicted block has successfully been written to host
[511:0] dataBlk2Cache	O	Memory Stage	The bus for the data block that was requested
dataBlk2CacheValid	O	Memory Stage	A signal to let the memory stage know the data on the bus is valid
accelWrBlkDone	O	Accelerator	Signals to the accelerator buffer

			that the block it shifted out has been written to host
accelRdBlkDone	O	Accelerator	Signals to the accelerator that the bus data is valid
[511:0] accelBlk2Buffer	O	Accelerator	A block of data being shifted out of the buffer to host memory
transformComplete	O	Accelerator	Signals to the accelerator once the last 8kB chunk of signal data has been transformed
[1:0] op	O	Mem Controller	Signals to the memory controller if there is a READ, WRITE, or IDLE request
common_data_bus_out	O	Mem Controller	The data being written to host memory
io_addr	O	Mem Controller	The address to read/write data in memory

The Arbiter communicates to host memory through the Memory Controller that was provided for us. The Arbiter abstracts requests so that the memory controller just needs an address and an op code to read/write from memory.

4. Implementation

Due to time constraints, synthesis was not completed. We spent most of our time trying to debug our algorithm for the FFT. With a completed FFT and simulation results that would have produced an expected output, synthesis would have been done.

5. Simulation Results

5.1 CPU

The CPU was tested in a bottom up method. The ALU, register file, and control unit were among the first modules that were tested via simulation. Next the caches were integrated into the fetch and memory stages and each were tested with the integration (more on this in the memory section). The cpu was then tested as a whole with accelerator and memory arbiter signals “spoofed” in the test bench. The test bench was written in a way to show that all outputs happened as expected as well as to show that the processor would run properly when it would be integrated with other modules. To do this, instruction blocks (512 bits or 16 instructions) were sent into the CPU at a time.

```
mcInstrIn = {{6{32'h00000000}}, // HALT will be skipped due to JR
             32'h28850000, // JR $1, 327680 (Jump to addy 'h00005000)
             32'h10000900, // STARTF signum (4), filter(1)
             32'h10000500, // STARTF signum (2), filter (1)
             32'hF8000600, // LOADF signum (3)
             32'h8B200000, // LD R4 <- MEM [R6 + 0 h'10000000] R4(h'1002)
             32'h83280000, // ST Mem[R6 + 0 (h'10000000)] <- R5 ('h1002)
             32'h93000000, // SLBI R6 zero filled so R6 = h'10000000
             32'h43280002, // ADDI R5 ('h1002) <- R6('h1000) + ('h02)
             32'hA3001000, // LBI R6 <- 'h00001000;
             32'h10000200}; // STARTF signum(1), filter (0)
```

Figure 12: First block of instructions sent into the CPU to be executed.

The first block of instructions started off with a startF instruction. This would demonstrate that the CPU would first have to wait for the memory arbiter to retrieve the 16 instructions from host memory due to a iCache miss. The CPU would get that first instruction once mcInstrValid was set, which was again a spoofed control signal in the test bench. Once the first startF signal was received, the

fftCalculating bit was set high afterwards to signify that the accelerator was calculating. This was to show that the processor could keep executing instructions that didn't conflict with FFT data regions.

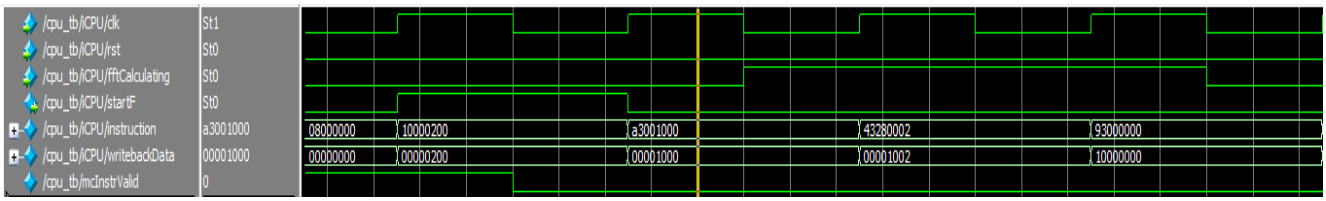


Figure 13: Nop instruction since cache Miss, then StartF, LBI, ADDI, and SLBI. It also shows when the fftCalculating was set and reset as well as mcInstrValid to show when the instructions were good to come into the iCache.

Next, a store and load instruction were sent to the processor. This was to show that they would get a dCache miss and have to retrieve the data from the arbiter. The testbench waited multiple cycles to simulate waiting for the arbiter then set mcDataValid high to show the host memory data can be written to the cache. The load instruction will then get a hit next since that block is not in the cache and is valid.

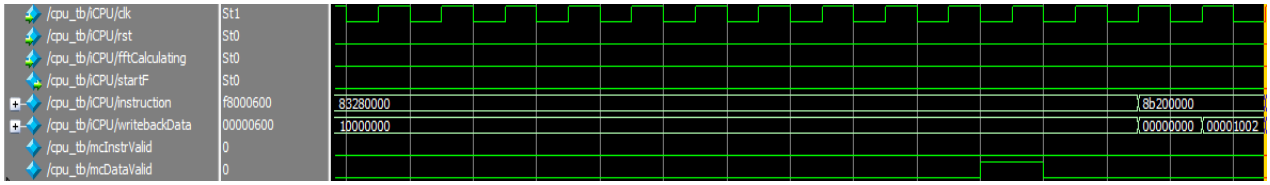


Figure 14: Shows the CPU stalling on the store instruction while it waits for mcDataValid signal. It then takes a few clock cycles to go through the state machine for the cache. Afterwards, the load instruction comes in and takes only two cycles to go through the cache's state machine as it is a hit in the cache.

The next few instructions involved more accelerator functionality. A LoadF instruction was sent to show that if the next startF seen had the filtering bit high, an exception would not occur since the filter was now loaded. This is exactly what the next startF has in its encoding. After the first startF, the testbench set fftCalculating high and another startF signal was sent. This is to show that the control signal startF for the accelerator would not go high as the accelerator was still calculating. As soon as the fftCalculating signal went low, the second startF would set the control signal and be processed.

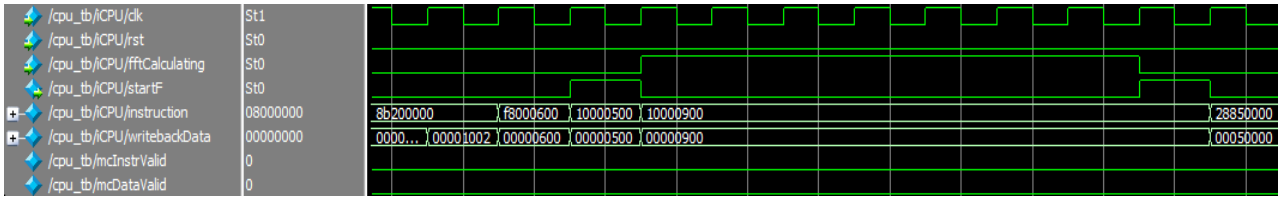


Figure 15: Shows the previous load instruction, then loadF followed by two startFs. The second startF does not set the control signal until the accelerator is done. After that control signal was set the processor correctly moved to the JR instruction.

After the first block of instructions, the next blocks sent just tested the general purpose part of the CPU. It tested every single instruction in the ISA with three more instruction blocks. All control signals were checked as well as checking that the PC properly jumped to different addresses if there was a jump or branch. As this wasn't the core purpose of the design, not much detail is needed to go into here. All general purpose instructions executed as expected.

```
mcInstrIn = {32'hda288000, //AND R1 <- R4 & ~R5 so R1 should get 0
             32'h599bfffd, //ANDNI R3 <- R3 & 3FFFD so R3 gets 0002
             32'h50985324, //XORI R3 <- R1 ^ 5324 should give R3 4326
             32'h89080000, //LD R1 <- MEM[R2] so R1 should get 1002
             32'h89080001, //LD the R1 <- MEM[R2 + 1] so R1 should get 0
             32'h99200A40, //STU MEM[10001A32] <- R4 (1002) R2 <- 10001A32
             32'hd3290000, //XOR R2 <- R6(10000000) ^ R5(1002) (R2 gets 10001002)
             32'he2288000, //SEQ R1 <- 1 cuz R5 and R4 equal (will get skipped becuase of JAL isntruction)
             32'h30000001, //JAL over 1 instruction but save address of the next instruction in R15 = 50008 (J instruction goes here)
             32'heaa08000, //SLT R1 <- 0 cuz R5 !< R4 (will get skipped because of the J instruction)
             32'h20000001, //Jump over one instruction
             32'heaa08000, //SLT R1 <- 0 cuz R5 !< R4 addr= 50004
             32'he2288000, //SEQ R1 <- 1 cuz R5 and R4 equal
             32'hce5d0000, //SUB R10 <- R11 - R12 R10 gets -5 (FFFB)
             32'h4e58000F, //SUBI R11 <- R12 15 - 10 (5)
             32'ha600000a //LBI R12 <- 10
    };
```

```
mcInstrIn = {{13{32'h0}}, //HALT, shouldn't get here because of branch
             32'h6a80138D,   //BNEZ R1 (R5 is 1) so branch to currAddr + 138D addrInstr = 0050012 (pcplus1 + offset)
             32'h62801390, //BEQZ R1 (R5 is 1) so no branch happens (addrInstr = 00050011)
             32'hf22a8000  //SLE R5 should get 1 since at this point R5 = 1002, R4 = 1002
    };
```

```
mcInstrIn = {{10{32'h0}}, //HALT, final instrcution
            32'h79800001, //should get skipped by the JALR
            32'h388513a6,  //JALR R1 + 1 (so skip 1 instrcution), R15 <- 513a6 (next instr addr) skips 1
            32'h465800c0, //addi R11 <- R12 + CA so R11 gets CA (gets here from bgez) instr addr = 513a3
            32'h00000000, //HALT, gets skipped because of BGEZ
            32'h79800001, //BGEZ R3, 1 (R3 greater than zero so should branch, skips 1 instruction)
            32'h710001f4 //BLTZ R2 (R2 greater than zero so shouldn't branch)
    };
```

Figures 16-18: Shows the final three instruction blocks sent to the CPU, which all executed as expected.

5.1.1 CPU Exceptions

The CPU also had exceptions that made the processor halt. The exceptions made it hard to do randomized stimulus for the overall CPU since randomizing could cause an exception to be raised, halting the CPU and causing no more random inputs to be tested. The exceptions were all tested in their own testbench and functioned as expected. More details can be found in the github repo at the bottom of the report. It was all simple combinational logic checking for each exception in its respective stage of the processor which then raised the exception and wrote the error code to the cause register.

5.2 Accelerator

The accelerator was tested in a bottom up fashion. The first of the tests were for simple functionality tests for the butterfly unit. I put in test data and tested for expected results calculated in software. The next test was for the address generator. To test this unit, it was put through all 10 stages of the FFT and the expected indices for each cycle in the stages were compared to the actual results.

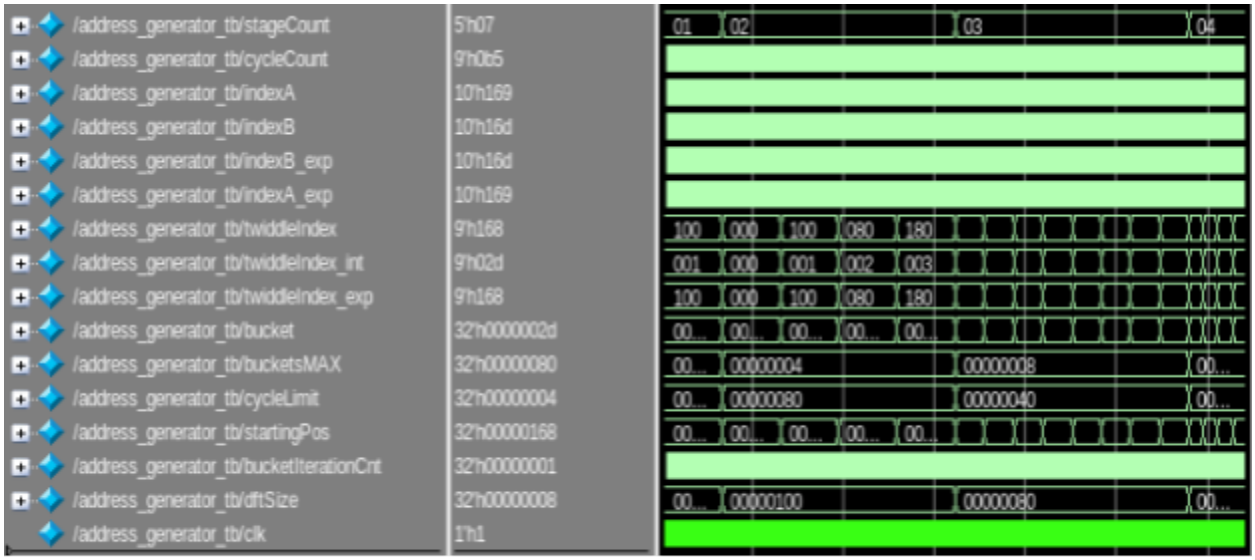


Figure 19: The waveform from the address generator testbench

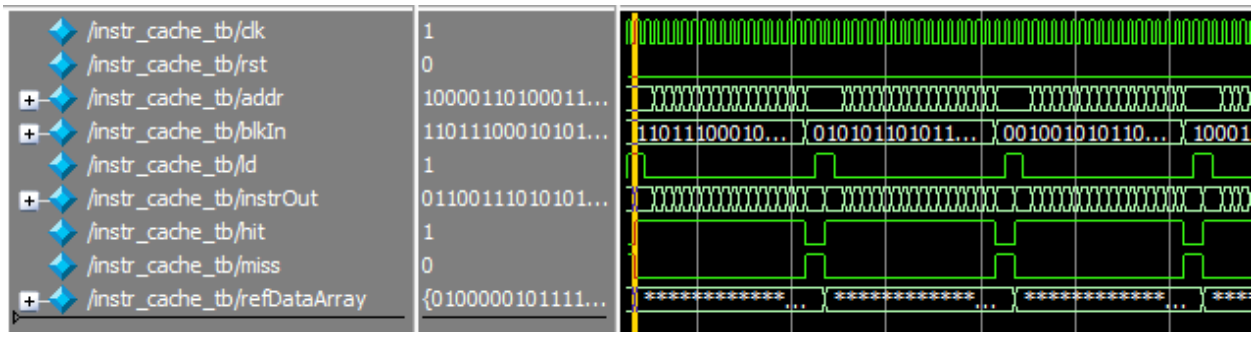
With the address generator and the butterfly unit, we were able to implement a mock FFT in a testbench with fake twiddle ROM and fake signal RAM. The fake twiddle ROM was loaded with twiddle factors generated from a MATLAB script and put into the testbench via the `$readmemh` function. The same goes for the signal, it was generated in MATLAB and cut up into a 1024 point sample, and loaded into the test bench. This testbench for testing a mock FFT proved that our algorithm was incorrect after heavily testing both the address generator and the butterfly unit and making sure they had no flaws.

With not a lot of time left, we tested the full functionality of the accelerator with all components besides the control unit (the testbench acting as the control unit). This testbench proved that all components worked as planned. After this test, the fifos and control unit were combined into a full accelerator module. There were some signal connectivity issues that were shown in our full accelerator testbench, but they were easy fixes and came to show that integration went really smoothly.

5.3 Memory

The memory systems were also tested in a bottom up fashion. Each cache was individually tested, then integrated into the fetch or memory stage and tested with just those two modules. The

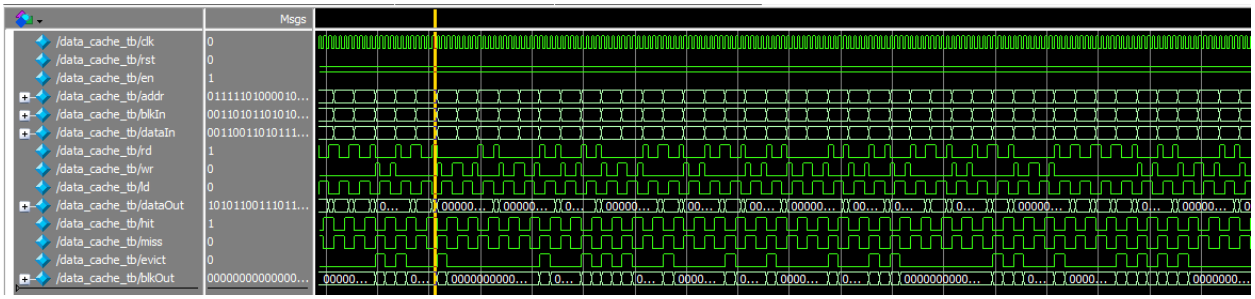
arbiter was then developed and tested alone. Next the buffers were implemented and tested together. Finally all parts of the design were tested together in our integration phase.




```
TEST 16375 |  READ | ADDR: b2d9b083
TEST 16376 |  READ | ADDR: 2d8089ec
TEST 16377 |  READ | ADDR: 06e22a69
TEST 16378 |  READ | ADDR: 7cd4ab1b
TEST 16379 |  READ | ADDR: f5c5e578
TEST 16380 |  READ | ADDR: 1102c1a3
TEST 16381 |  READ | ADDR: 70d9932e
TEST 16382 |  READ | ADDR: a6eec290
TEST 16383 |  READ | ADDR: 17d5ed95
TEST 16384 |  READ | ADDR: f85fde79
TEST PASSED
```

Figures 20-21: A randomized sequence of reads occurs, if a block isn't present in the cache, the new block gets loaded, and then all instruction words from that block are read from and checked with reference memory.

As shown above, the instruction cache was tested with a fully randomized self-checking test bench. There are 16384 iterations that generate a random address from memory, which loads a block from host memory, and then attempts to read the 16 instruction words from that line.



```
TEST 16380 |  READ | ADDR: 59fb2aec
READ EVICT
TEST 16381 | WRITE | ADDR: ad64b2a1
WRITE EVICT
TEST 16382 | WRITE | ADDR: 3047d1ea
TEST 16383 | WRITE | ADDR: 3fcac960
WRITE EVICT
TEST 16384 |  READ | ADDR: 441663a3
READ EVICT
TEST PASSED
```

Figures 22-23: Randomized tests are generated, either reading or writing from a block. If there is a miss, the cache will let us know if an eviction has occurred. If so, the data sent out from the cache is checked against our reference memory to ensure that the changes are present. The new block is then put into the cache.

The data cache was tested in a similar manner, randomized addresses were sent to the cache, and upon a miss it would load in the respective block from a reference data array, and then those 16 blocks were read from that line, making sure that the result matched with memory. There was also the chance of an eviction or a write, in which case the evicted block was stored back into the reference memory, and checked to make sure that it had updated data.

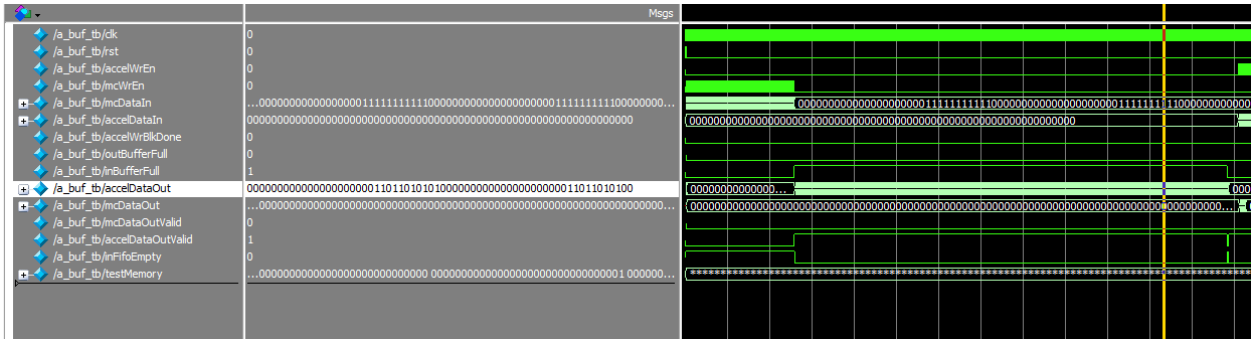


Figure 24: A-Buffer shifts in data during the first 128 clock cycles, then as inBufferFull goes high, the buffer will start shifting out data points for the next 1024 clock cycles.

For the A-Buffer test bench, visual inspection of the buffers were used because of their design simplicity. A reference memory bank was created, and the buffer shifted in 512b at a time incrementally. After the buffer is full, the accelDataOut bus gets shifted each 64b data point per clock cycle. Ascending values were used to make visual inspection easier. Similarly, with the Out Buffer, 64b data points were

shifted in until the Buffer was full, then 512b values were shifted out to the mcDataOut bus. This was also verified using ascending data values and visual inspection.

The arbiter was also tested using visual inspection, it was firstly tested with single requests from the instruction cache. Once that was implemented correctly, we tested it with multiple simultaneous data and instruction cache requests from different locations in memory. Again, a spoofed “host memory” array was used with ascending data to make it easy to debug. The data cache also modified some data and eviction requests were handled. Then we phased in the accelerator buffer request to ensure that a full 8kB region of data was transferred to the buffer uninterrupted, and then data or instruction requests could be granted.

5.4 Top Level Processor

The top level processor with the accelerator, cpu, and memory arbiter all integrated was simulated via an integration test bench. The memory control signals were “spoofed” to simulate its functionality. The instruction memory was loaded with an instruction block to simulate the core functionality of the integration with the accelerator.

```
instrMemory[0] = 32'hA3003000; // LBI R6 <- 'h00003000;
instrMemory[1] = 32'h43280002; // ADDI R5 ('h3002) <- R6('h3000) + ('h02)
instrMemory[2] = 32'h93000000; // SLBI R6 zero filled so R6 = h'30000000
instrMemory[3] = 32'h83280000; // ST Mem[R6 + 0 ('h30000000)] <- R5 ('h3002) //should cause a DMA request
instrMemory[4] = 32'h8B200000; // LD R4 <- MEM [R6 + 0 'h30000000] R4('h3002) //should be a hit
instrMemory[5] = 32'h10000000; // STARTF signum(0), filter (0)*/ //should cause accelerator request
instrMemory[6] = 32'ha600000a; // LBI R12 <- 10 load 10 should work while accelerator is doing startF
instrMemory[7] = 32'h10000000; // STARTF signum (0), filter (0) gets stalled while accelerator does work
instrMemory[8] = 32'h00000000; //HALT (nothing after this gets executed)
```

Figure 25: The instructions that were simulated in the integration test bench.

First, we tested that some general purpose instructions worked in the cpu. We then did a store instruction followed by a load to test that the memory arbiter correctly put in data to the data cache. A startF instruction was then issued to make sure that the accelerator would get the control signal to start calculating. We then issued another general purpose instruction to show the CPU can continue doing

work while the accelerator is working. Since this was tested extensively in the CPU testbench, we didn't test more than one instruction because if one worked we were confident that more would work as long as they didn't conflict with accelerator data. Lastly, we executed another startF to show that the CPU would stall on this instruction until the accelerator was done calculating on the first startF instruction. The functionality all proceeded as expected and is shown in the waveform figures below.

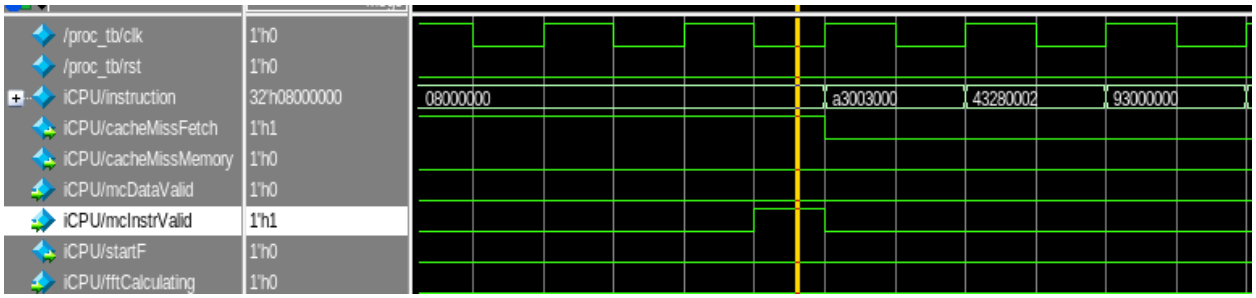


Figure 26: Shows the instruction miss until instrValid goes high then the general purpose instructions execute as normal.

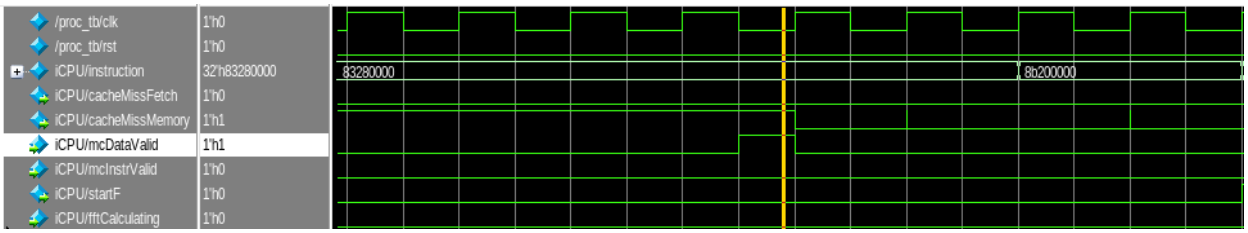


Figure 27: Shows the data cache miss on the store then proceeding to the load after the data has been written to the cache.

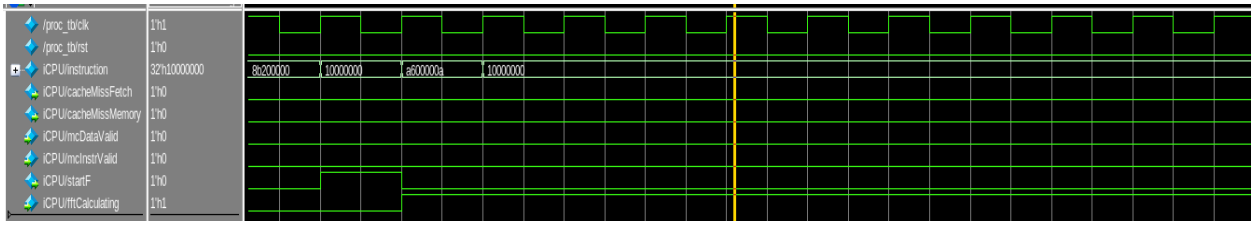


Figure 28: Shows the first startF setting fftCaclulating high. Then the LBI instruction executes while the accelerator works. Lastly, the startF instruction is stalled in the processor while the first startF calculates (not setting startF control signal yet).

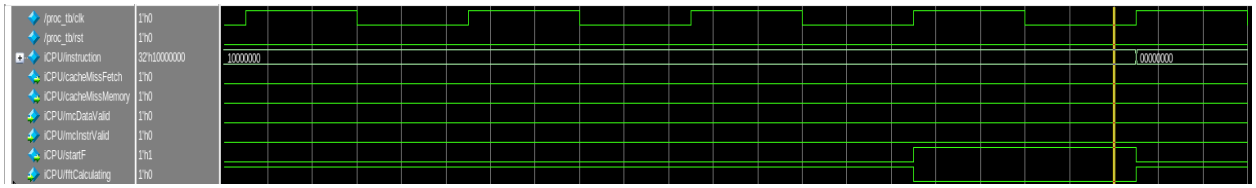


Figure 29: Shows the second startF starting after the first clears fftCalculating. It then moves onto the last halt and the second startF gets processed in the background.

6. Demo

6.1 Software vs Hardware results

Our software demo consisted of a comparison of our DFT algorithm implemented in software vs a hardware implementation. The software results were generated within a matlab file, with results displaying the frequency domain of the original signal. The hardware results were generated from the testbench running through our System Verilog code. These results were then written to a text file. From there, that text file was parsed by a C++ script, and entered into matlab. Once in Matlab, the hardware results were then able to be visualized in the exact same format of the software results. What was found was that the hardware results exactly matched the software results.

The only discrepancy regarding the validity of our results is that there was a slight error in our FFT algorithm, and both the software and hardware implementations contained this error. Our team worked hard to find this small error and found after a long period of time that we were resorting to guessing and checking. We decided it would be best for the project to just focus on integrating all functionalities, as the algorithm *had* been implemented correctly in hardware, though it was not a DFT.

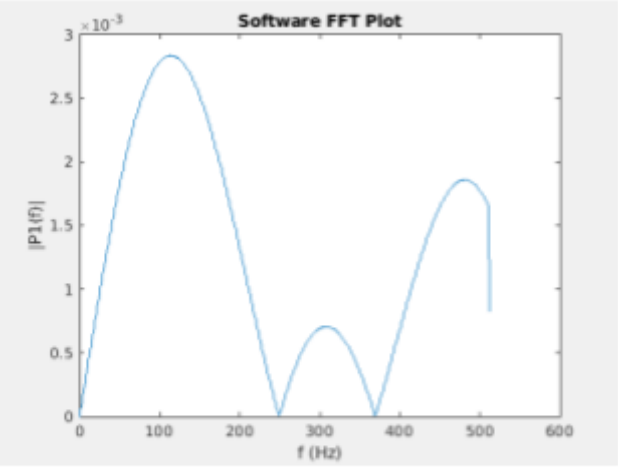


Figure 30: Output from the software FFT Algorithm we came up with

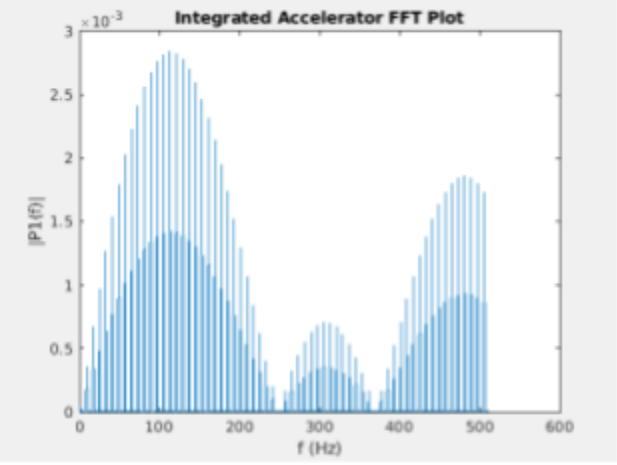


Figure 31: Output from our hardware implementation

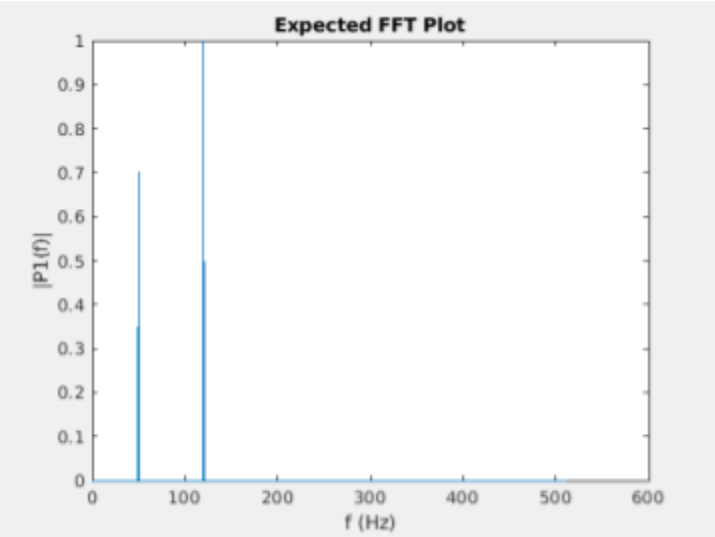


Figure 32: Shows the expected output of the FFT on the test signal

7. Project Status

7.1 Completeness of Design

Our final design is not what we expected it to be at the start of the process due to time constraints. We have a fully functioning general purpose processor that should be able to work with the accelerator design laid out. However, we were not able to get a fully functional exception handling design. As of now, the processor just treats an exception like a halt and will stop executing instructions when one is raised.

The memory caches were all tested and worked as expected. The same goes for the accelerator buffers as those are shifting in and out data correctly. In the Memory Arbiter, it was supposed to be able to handle up to 2GB in signal data, or 262,144 separate 8kB signals. Because we were not able to synthesize, and due to other time constraints, it was impractical to implement this how we wished. Instead we added support for 1 8kb signal that was loaded into a predetermined memory location. The structure of the sigNum table, and functionality, are implemented, but hardcoded to only read a signal from the first block of our Accelerator data section. This was done to simplify integration testing and to ensure that we could verify our Accelerator in the simplest manner possible. The next step would be to read in signal base and bounds from the static data section in host memory, and fill the table. Then the accelerator would know if there was another chunk of signal data to transform or if it was done with the full signal.

As for the accelerator, we could not get the algorithm flushed out properly to perform an FFT. This is where we spent a lot of our time and in hindsight, we should have written the algorithm in software earlier. The bright side is that the algorithm in hardware matches the software implementation of the algorithm we flushed out, so we implemented the wrong algorithm correctly. Since most of our

time went to trying to get the normal FFT working, we were unable to get the accelerator to do an inverse FFT or add a filtering block in.

7.2 Integration

7.2.1 Hardware

Our integration phase went very smoothly. Because we constructed architecture documents ahead of time, each of the group members were able to follow their specs, and the full functionality of the design worked almost right away. There were a few discrepancies such as if signals should hold high for just a clock cycle, or stay high until they were properly taken care of. These are some things we did not mention or specify in our architecture documentation. Another thing that we struggled with was keeping track of signal names. Because we used a lot of simple names in sub-modules, such as read, write, or enable, it became a difficult task to route them to the correct place in our top-level module. Instead, we went back and renamed a lot of the signals to be more semantically descriptive.

7.2.2 Software

The software was originally intended to be simulated by a single bash script enabling the following: assembly to binary data, assembly instruction input through direct memory access, capture output from software and hardware, several python and matlab scripts to create a comparative FFT of the input signal. Due to time constraints, such a streamlined file was too large of an obstacle for the semester.

7.2.2.1 Assembler: Testing & Integration

The assembler was created using the 'class' library. The assembler was then refactored to fit our instruction set. Our software is tested by the functionality of the accelerator itself; the instructions fed to the processor are generated by the assembler. The assembler was to be tested and used by the 'cload' functionality, placing instructions into the virtual memory of our integrated processor after being assembled by a C++ program. That step was not reached within the constraints of the assignment. Our testing strategy was to generate a list containing every instruction using the assembler, then feed those instructions into a HDL testbench. With that we can execute instructions and check that our simulation memory has been properly updated, with the correct instruction format. Exceptions could also be tested in this format, as an illegal instruction state can easily be obtained once the instructions can be generated.

7.2.2.2 MATLAB and C++ testing scripts/programs

In order to develop our test data, as well as convert it into a format readable by the testbench, MATLAB scripts had to be developed, as well as some C++ programs. The MATLAB scripts were used to create test signals. The test signal we used was a sinusoid with frequencies of 50Hz and 120Hz. This signal was outputted from the script into a txt file.

Once the signal was in the file, a C++ program was used to convert the decimal data into hexadecimal data, which could then be read by the Modelsim program which ran our hardware testbenches. The Modelsim program then outputs the resulting data from the FFT into a txt file, which, again, was in hexadecimal. This data was then converted back into decimal format that could be easily readable to MATLAB. MATLAB then took in this decimal data from the testbench and C++ program, read it, and plotted the resulting data in the frequency domain.

7.3 System Test

Our system test was the series of tests contained in our proc hierarchy testbench. It is actually the same as the top level processor simulation, outlined in section 5.4 of the report. The system worked as expected with everything integrated. However, as mentioned before, the algorithm was incorrect so our results were not the expected FFT.

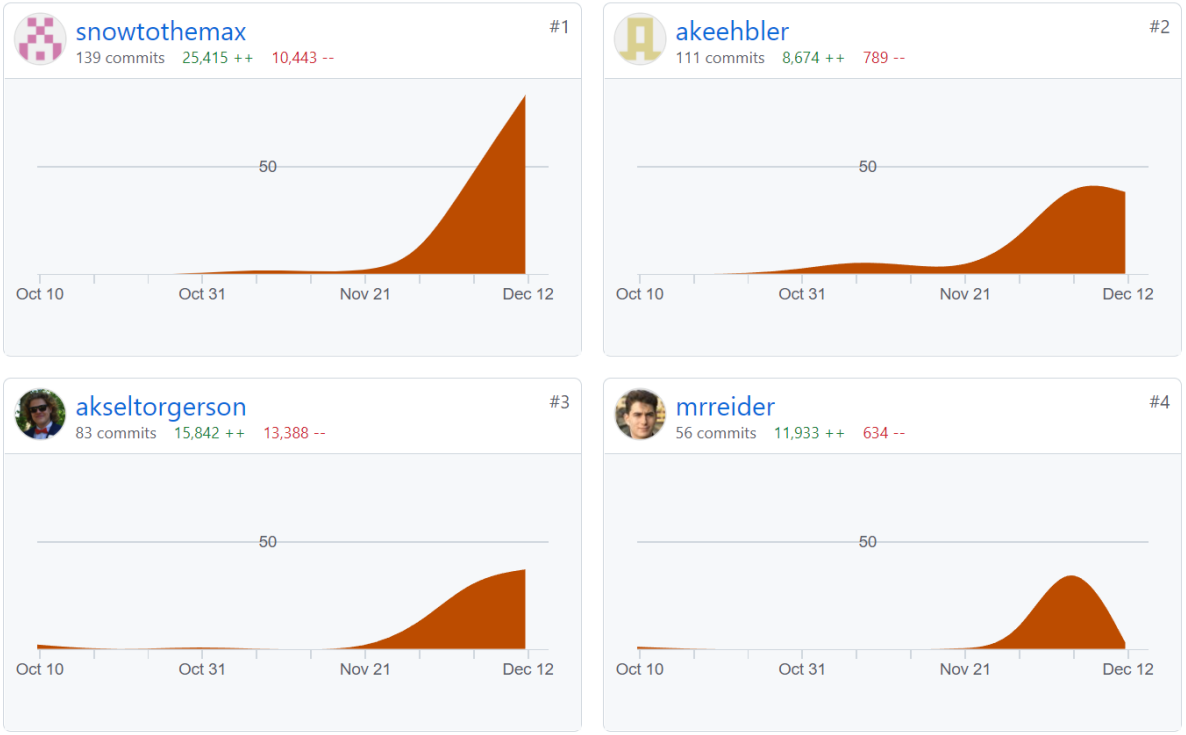
8. Team Contribution

Alec Keehbler: CPU, ISA, Integration, and testing (CPU and top level processor)

Max Johnson: Accelerator, Integration, and testing (accelerator and output data for demo)

Aksel Torgerson: Caches, Memory Arbiter, A Buffers, Integration, and testing (memory and top level processor)

Reid Brostoff: Assembler, Software, consulted on Exceptions and A Buffers



Github repository link: https://github.com/akseltorgerson/ECE554-Capstone

Signed:

Alec Keehbler
Aksel Torgerson
Max Johsnon
Reid Brostoff