The WI-23 General Purpose Graphing Calculator – Final Project Report

Wisconsin Instruments

Elan Graupe, Aidan McEllistrem, Ayaz Franz, Julien De Castelnau, and Madhav Rathi

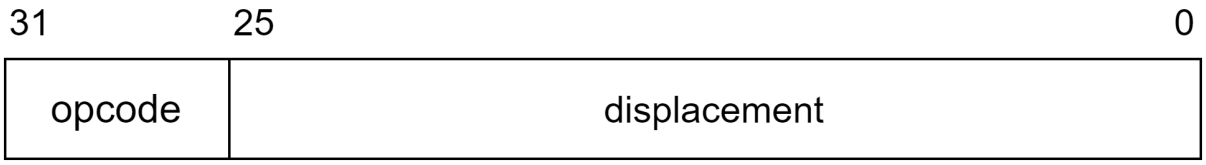
Table of Contents

1. Instruction Set Architecture (ISA)

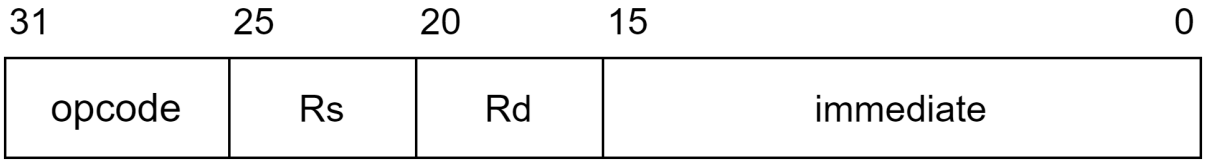
WI-23 is a full 32-bit architecture with a 32-bit datapath and 32-bit instructions with floating point support. All floating-point operations use IEEE-754 32-bit single precision binary representation.

1.1 Instruction Formats

J-Format

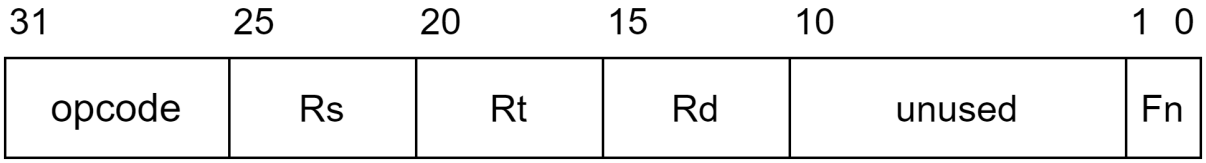


I-format



Some I-format instructions do not use Rd. In this case, these bits are treated as don't-care.

R-format



Some R-format instructions do not use Rt. In this case, these bits are treated as don't-care.

Others

Some special instructions do not use any registers or immediates. In this case, only the opcode is read, all other bits are don't-care.

1.2 Instructions

Format	Opcode	Fn	Syntax	Semantics	Other Comments
Other	000000	XX	HALT	Cease instruction issue. Assert halt signal at top-level.	Used for simulation.
Other	000001	XX	NOP	None	
I-format	000011	XX	LDCR Rd, Rs, imm	Rd <- IMem[Rs + I(sign ext.)]	Load word from instruction memory
I-format	001000	XX	ADDI Rd, Rs, imm	Rd <- Rs + I(sign ext.)	
I-format	001001	XX	SUBI Rd, Rs, imm	Rd <- I(sign ext.) - Rs	
I-format	001010	XX	XORI Rd, Rs, imm	Rd <- Rs XOR I(zero ext.)	
I-format	001011	XX	ANDNI Rd, Rs, imm	Rd <- Rs AND ~I(zero ext.)	
I-format	010100	XX	ROLI Rd, Rs, imm	Rd <- Rs <<(rotate) I(lowest 5 bits)	
I-format	010101	XX	SLLI Rd, Rs, imm	Rd <- Rs << I(lowest 5 bits)	
I-format	010110	XX	RORI Rd, Rs, imm	Rd <- Rs >>(rotate) I(lowest 5 bits)	
I-format	010111	XX	SRLI Rd, Rs, imm	Rd <- Rs >> I(lowest 5 bits)	
I-format	010000	XX	ST Rd, Rs, imm	Mem[Rs + I(sign ext.)] <- Rd	Store word to data memory
I-format	010001	XX	LD Rd, Rs, imm	Rd <- Mem[Rs + I(sign ext.)]	Load word from data memory
I-format	010011	XX	STU Rd, Rs, imm	Mem[Rs + I(sign ext.)] <- Rd Rs <- Rs + I(sign ext.)	

I-format	011001	XX	ANDI Rd, Rs, imm	Rd <- Rs AND I(zero ext.)	
I-format	111001	XX	ORI Rd, Rs, imm	Rd ← Rs OR I (zero ext.)	
R-format	011011	00	ADD Rd, Rs, Rt	Rd <- Rs + Rt	
R-format	011011	01	SUB Rd, Rs, Rt	Rd <- Rt - Rs	
R-format	011011	10	XOR Rd, Rs, Rt	Rd <- Rs XOR Rt	
R-format	011011	11	ANDN Rd, Rs, Rt	Rd <- Rs AND ~Rt	
R-format	011010	00	ROL Rd, Rs, Rt	Rd <- Rs << (rotate) Rt (lowest 5 bits)	
R-format	011010	01	SLL Rd, Rs, Rt	Rd <- Rs << Rt (lowest 5 bits)	
R-format	011010	10	ROR Rd, Rs, Rt	Rd <- Rs >> (rotate) Rt (lowest 5 bits)	
R-format	011010	11	SRL Rd, Rs, Rt	Rd <- Rs >> Rt (lowest 5 bits)	
R-format	011100	XX	SEQ Rd, Rs, Rt	Rd <- (Rs == Rt) ? 1 : 0	
R-format	011101	00	SLT Rd, Rs, Rt	Rd <- (Rs < Rt) ? 1 : 0	Signed
R-format	011101	01	SLTU Rd, Rs, Rt	Rd <- (Rs < Rt) ? 1 : 0	Unsigned
R-format	011110	00	SLE Rd, Rs, Rt	Rd <- (Rs <= Rt) ? 1 : 0	Signed
R-format	011110	01	SLEU Rd, Rs, Rt	Rd <- (Rs <= Rt) ? 1 : 0	Unsigned
R-format	011111	00	AND Rd, Rs, Rt	Rd ← Rs AND Rt	
R-format	011111	01	OR Rd, Rs, Rt	Rd <- Rs OR Rt	
I-format	001100	XX	BEQZ Rs, imm	if (Rs == 0) then PC <- PC + 2 + I(sign ext.)	Rd field unused.
I-format	001101	XX	BNEZ Rs, imm	if (Rs != 0) then PC <- PC + 2 + I(sign ext.)	Rd field unused.
I-format	001110	XX	BLTZ Rs, imm	if (Rs < 0) then PC <- PC + 2 + I(sign ext.)	Rd field unused.
I-format	001111	XX	BGEZ Rs, imm	if (Rs >= 0) then PC <- PC + 2 + I(sign ext.)	Rd field unused.
I-format	011000	XX	LBI Rs, imm	Rs <- I(sign ext.)	Rd field unused.
I-format	010010	XX	SLBI Rs, imm	Rs <- (Rs << 16) | I(zero ext.)	Rd field unused.
J-format	000100	XX	J disp	PC <- PC + 2 + D(sign ext.)	26-bit displacement.

I-format	000101	XX	JR Rs, imm	PC <- Rs + I(sign ext.)	Rd field unused.
J-format	000110	XX	JAL disp	R30 <- PC + 2 PC <- PC + 2 + D(sign ext.)	26-bit displacement.
I-format	000111	XX	JALR Rs, imm	R30 <- PC + 2 PC <- Rs + I(sign ext.)	
R-format	111011	00	FADD Fd, Fs, Ft	Fd <- Fs + Ft	Overflow sets Fd to INFINITY (+ or - depending on which direction the operation overflowed).
R-format	111011	01	FSUB Fd, Fs, Ft	Fd <- Ft - Fs	
R-format	111011	10	FMUL Fd, Fs, Ft	Fd <- Fs * Ft	Underflow results in a signed 0 using the sign of Fs.
R-format	111100	XX	FEQ Fd, Fs, Ft	Fd <- (Fs == Ft) ? 1 : 0	Term +0 is not equal to -0. +INFINITY is not equal to -INFINITY. NaN is always != anything (including itself)
R-format	111110	XX	FLE Fd, Fs, Ft	Fd <- (Fs <= Ft) ? 1 : 0	
R-format	111111	XX	FLT Fd, Fs, Ft	Fd <- (Fs < Ft) ? 1 : 0	
R-format	100001	XX	FCVTI Fd, Rs	Fd <- (float)Rs	Convert Int to FP: that is, the integer representation of the number in the Rs register is converted to an IEEE-754 compliant number.

R-format	100000	XX	ICVTF Rd, Fs	Rd <- (int)Fs	Inverse of FCVTI.
R-format	100011	XX	FMOVI Fd, Rs	Fd <- Rs	Move an integer register to the floating point regfile.
R-format	100010	XX	IMOVF Rd, Fs	Rd <- Fs	Inverse of FMOVI
I-format	110000	XX	FST Fd, Rs, imm	Mem[Rs + I(sign ext.)] <- Fs	
I-format	110001	XX	FLD Fd, Rs, imm	Fd <- Mem[Rs + I(sign ext.)]	
I-format	110010	XX	STB Rd, Rs, imm	Mem[Rs + I(sign ext.)] <- Rd	Store Byte to DMEM
I-format	110011	XX	LDB Rd, Rs, imm	Rd <- Mem[Rs + I(sign ext.)]	Load Byte from DMEM
I-format	110100	XX	STH Rd, Rs, imm	Mem[Rs + I(sign ext.)] <- Rd	Store Half-Word to DMEM
I-format	110101	XX	LDH Rd, Rs, imm	Rd <- Mem[Rs + I(sign ext.)]	Load Half-Word from DMEM
I-format	101010	XX	XORSI Rd, Rs, imm	Rd <- Rs XOR I(sign ext.)	
I-format	110111	XX	SRAI Rd, Rs, imm	Rd <- Rs >>> I(lowest 5 bits)	Arithmetic shift right
R-format	111010	XX	SRA Rd, Rs, Rt	Rd <- Rs >>> Rt (lowest 5 bits)	Arithmetic shift right

1.3. Register Sets

Register	ABI name	Description
Special Purpose		
r30	ra	Return address is stored here by JAL-type instructions.
General Purpose		
Some might have fixed functions but do not affect hardware if changed.		

r29	sp	Stack pointer
r28	fp	Frame pointer
r0-r27, r31	r0-r27, r31	General-purpose registers.
General Purpose Floating Point		
f0-f31	f0-f31	General-purpose floating point registers.

1.4. Condition Codes

N/A - WI-23 does not use condition codes. All conditions are evaluated explicitly (using branches if equal/not equal and set instructions).

1.5. Addressing Modes

WI-23 is a load-store architecture which supports the following addressing modes:

1. Register base-immediate addressing: Used by load, store, jump (and link) register. Uses a register as a base and adds immediate.
2. PC-relative addressing: Used by unconditional jump/jump and link as well as branch instructions. Jump has room for a 26-bit displacement while branch is only 16.

1.6. Immediate

Imm16 : 16-bit immediate.

- Sign- or Zero- extended
- Used with all I-format instructions. Arithmetic, loads, stores, and branches.

Disp: 26-bit displacement

- Used for J-format instructions: Jump and Jump and Link.

Immediate value constructed in decode stage and used in the following stages.

1.6. Harvard vs Von Neumann

The WI-23 Graphing Calculator uses a Harvard architecture inherited from the WISC-SP13 architecture.

2. Hardware

The WI-23 General Purpose Graphing Calculator is a graphing calculator implemented via an FPGA, PS/2 keyboard input, and VGA output. It has similar utility to a TI-84 handheld graphing calculator, hence the name.

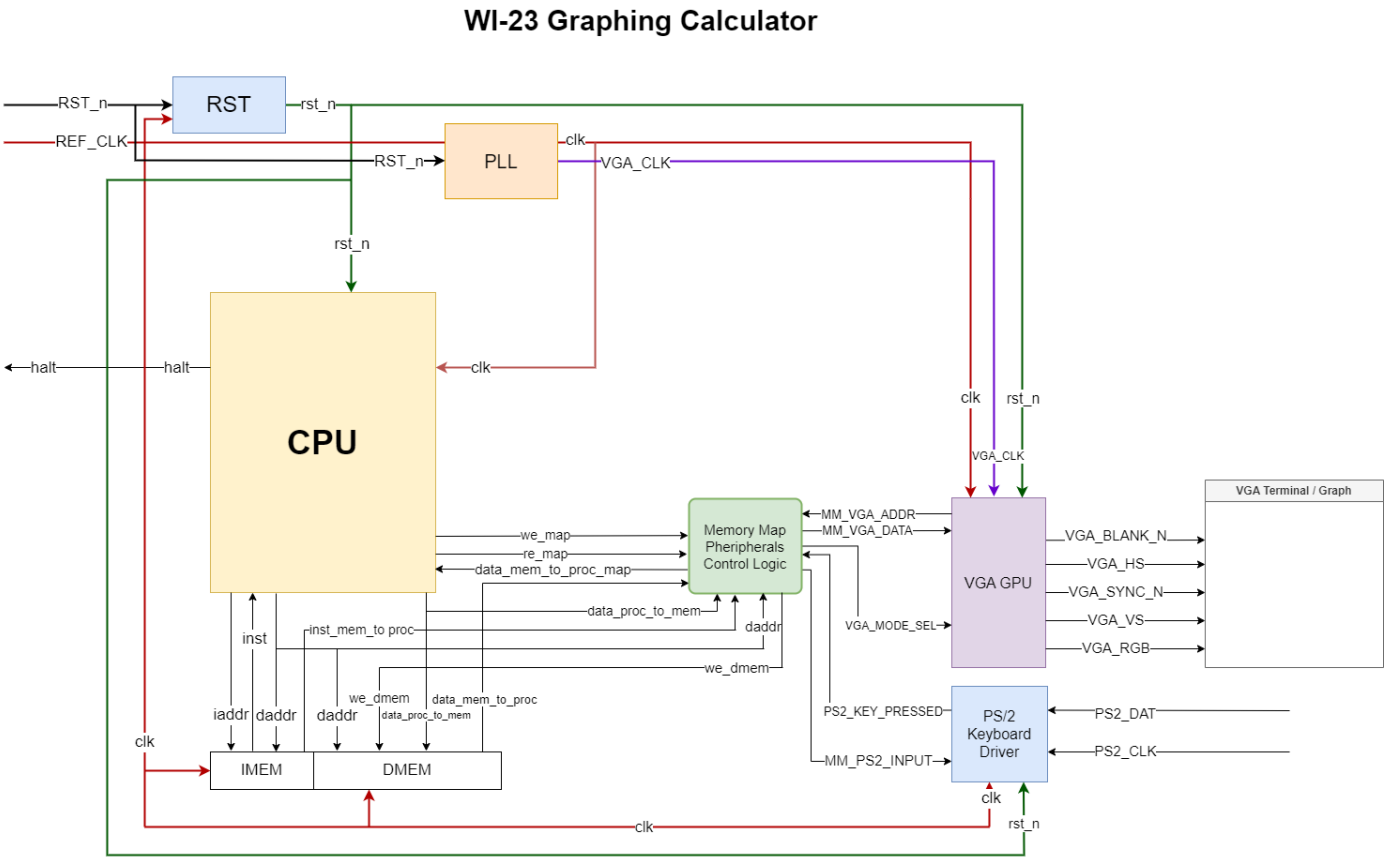
A wide range of floating-point math operations are provided for use by the graphing calculator REPL software (and supported by the underlying hardware), and will be graphed in an equation editor onto a VGA display.

The source code can be found at:
https://github.com/panther03/ECE-554-ClassProject/tree/main

2.1 Top-Level

The top-level blocks handle all the I/O and CPU operations. Each block performs the below mentioned functions -

1. CPU: General-purpose CPU which can handle integer and FP instructions. The CPU also interfaces with the memory mapped peripherals to read user inputs and output data on to the IO.
2. RST: Reset synchronizer which synchronizes the FPGA reset. The reset is active low.
3. VGA: The VGA GPU / driver is designed to both offer practical drawing options for a graphing calculator as well as minimal framebuffer memory usage. It supports a VGA text buffer.
4. PS/2 Keyboard: Interface to the PS/2 keyboard I/O.
5. Memory Map Peripheral Logic: Logic to collate and drive all memory map interface signals.



2.1.A. Interface

Signal	Input/Output	Width	Description
clk	Input	[0]	50MHz clock
RST_n	Input	[0]	FPGA Reset (Not Synchronized)
halt	Output	[0]	CPU encountered a Halt instruction. Used for debug.
RX	Input	[0]	UART RX line
TX	Input	[0]	UART TX line
VGA_CLK	input	[0]	Timing for the VGA unit.
VGA_BLANK_N	output	[0]	Asserted low between frames when nothing is being drawn.
VGA_HS	output	[0]	Active low horizontal sync

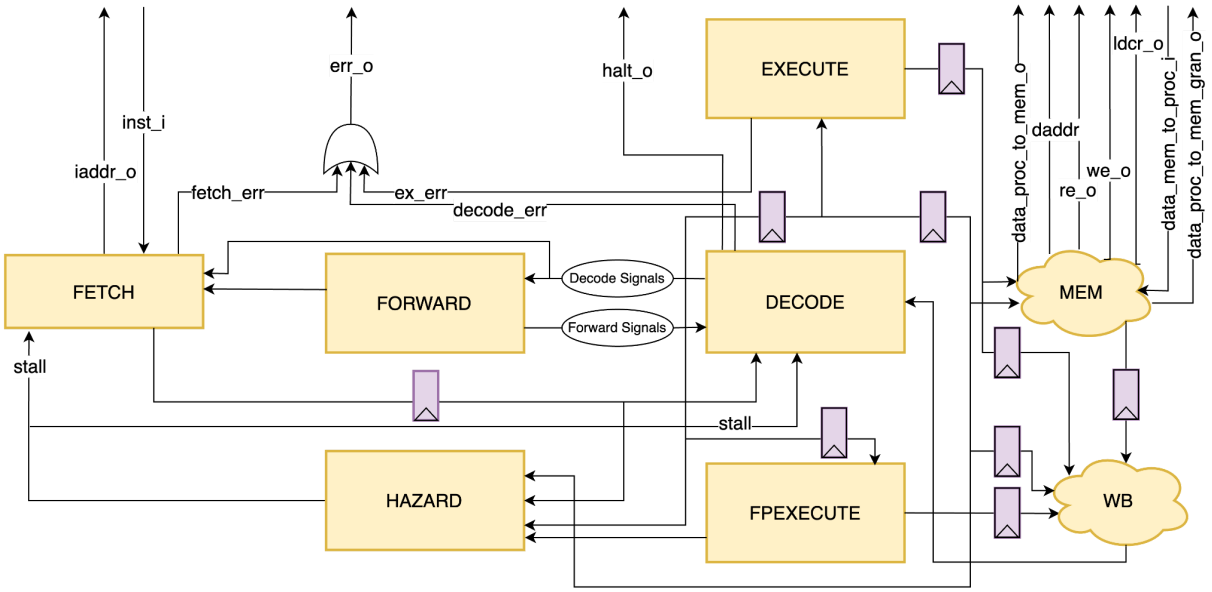
			for positioning the video signal.
VGA_SYNC_N	output	[0]	Controls V-sync.
VGA_VS	output	[0]	Active low vertical sync for positioning the video signal.
VGA_R, VGA_G, VGA_B	output	[7:0], [7:0], [7:0]	Red, green, and blue components for the VGA video signal.
PS2_DAT	input	[0]	Data from the PS/2 keyboard serial interface.
PS2_CLK	input	[0]	Clock from the PS/2 keyboard serial interface.

2.2. CPU

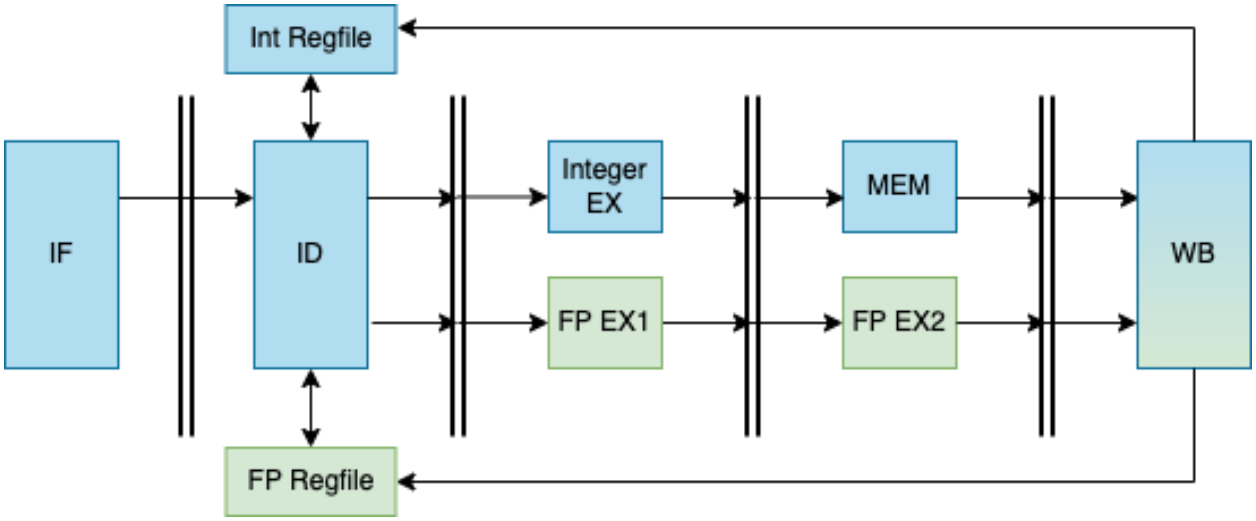
The CPU contains blocks and pipeline flops for the FETCH, DECODE, EXECUTE, FPEXECUTE, MEM and WB stages. It also has separate blocks to handle forwarding and hazard detection logic.

Block descriptions -

1. FETCH: Fetch block handles next PC calculation, flushing of speculative instructions and requesting a read from the instruction memory.
2. DECODE: Decodes the instruction, handles read/write to the register file and outputs control signals.
    a. Int Regfile: Dual-ported integer register file
    b. FP Regfile: Dual-ported FP register file
3. Integer EX: Performs integer ALU operations for all integer instructions and FP LD/ST instruction
4. FP EX: Performs FP ALU operations in 2-cycles for all FP instructions except FP LD/ST instruction
5. MEM: Assigns output data memory and memory map peripherals signals
6. WB: Muxes write data to the register files from integer and FP pipelines
7. HAZARD: Hazard detection unit stalls the pipeline if a hazard can’t be resolved through register forwarding
8. FORWARD: Forwarding unit forwards register values to EX and FEX stage from MEM and WB stage



The pipeline is as follows -



2.2.A. Interface

Signal	I/O	Width	Description
clk	Input	[0]	Clock
rst_n	Input	[0]	Active Low Reset
iaddr_o	Output	[31:0]	Instruction Memory address to read

ldcr_o	Output	[0]	Instruction Memory read for LDCR instruction
inst_i	Input	[31:0]	Instruction read from IM
daddr_o	Output	[31:0]	Data memory or memory map peripheral address
we_o	Output	[3:0]	Write enable to data memory and memory map peripherals
re_o	Output	[0]	Read enable to data memory and memory map peripherals
data_proc_to_mem_o	Output	[31:0]	Write data to data memory or memory map peripherals
data_proc_to_mem_gram_o	Output	[1:0]	Memory access granularity (00: Word, 01: Byte, 10: Half-word)
data_mem_to_proc_i	Input	[31:0]	Data from data memory or memory map peripherals
err_o	Output	[0]	Error in CPU
halt_o	Output	[0]	Halt seen

2.2.B. Registers

All pipeline latches.

2.2.1. Fetch

Fetch block handles next PC calculation, flushing of speculative instructions and requesting a read from the instruction memory.

2.2.1.A Registers

Register	**Width**	**Description**
PC	[31:0]	Program Counter

2.2.1.B Interface

Signal	I/O	Width	Description
clk	Input	[0]	Clock
rst_n	Input	[0]	Active Low Reset
reg1	Input	[31:0]	Read register for J type.
imm	Input	[31:0]	JType Immediate
ofs	Input	[31:0]	JType Offset
pc_inc_in	Input	[31:0]	New PC (PC+4 or PC when stall)
stall	Input	[0]	Stall in progress. Recirculate PC.
Halt	Input	[0]	Halt PC increment
Exc	Input	[0]	To be used when an interrupt is encountered.
JType	Input	[1:0]	JType[1] - PC Relative JType[0] - Immediate or Displacement Used to calculate the next PC.
CondOp	Input	[1:0]	Branch conditions. Used to calculate the next PC.
iaddr	Output	[31:0]	Output address to IM (word-aligned)
pc_inc_out	Output	[31:0]	Next PC (PC+4 or PC when halt)
flush	Output	[0]	Encountered taken branch/jump. Flush.
fetch_err	Output	[0]	Tied to zero.

2.2.2. Decode

Decode block performs the following functions:

1. Decode 32-bit instruction
2. Output control signals
3. Read and write the register file

2.2.2.A Registers

Register	Description
iRF.rf1	Integer register file. 32 registers.
iRF.rf2	Integer register file. 32 registers. Copy of rf1. Required to support dual-ported RF.
iFPRF.rf1	FP register file. 32 registers.
iFPRF.rf2	FP register file. 32 registers. Copy of rf1. Required to support dual-ported RF.

2.2.2.B Interface

Signal	I/O	Width	Description
clk	Input	[0]	Clock
rst_n	Input	[0]	Active low reset
Inst	Input	[31:0]	Instruction to decode
write_in	Input	[31:0]	Integer register write value
writesel	Input	[4:0]	Integer register write index
fp_write_in	Input	[31:0]	FP register write value
fp_writesel	Input	[4:0]	FP register write index
bypass_reg1	Input	[0]	Bypass integer register source A
bypass_reg2	Input	[0]	Bypass integer register source B
fp_bypass_reg1	Input	[0]	Bypass FP register source A
fp_bypass_reg2	Input	[0]	Bypass FP register source B

WB_RegWrite	Input	[0]	WB register write to integer register file
WB_FPRegWrite	Input	[0]	WB register write to FP register file
reg1	Output	[31:0]	Read register source A
reg2	Output	[31:0]	Read register source B
imm	Output	[31:0]	Extended immediate
ofs	Output	[31:0]	Extended displacement
fp_reg1	Output	[31:0]	Read FP register source A
fp_reg2	Output	[31:0]	Read FP register source B
RegWrite	Output	[0]	Control signal - Register Write
MemWrite	Output	[0]	Control signal - Memory Write
MemRead	Output	[0]	Control signal - Memory Read
InstFmt	Output	[1:0]	Control signal 00 - I-Format 2 01 - I-Format 1 10 - R-Format 11 - J-Format
MemToReg	Output	[0]	Control signal - Write to register from mem
AluSrc	Output	[0]	Control signal - Immediate value is a source
AluOp	Output	[4:0]	Control signal - Operation to perform in Alu
CondOp	Output	[1:0]	Branch conditions. Used to calculate the next PC.
JType	Output	[1:0]	JType[1] - PC Relative JType[0] - Immediate or Displacement
XtendSel	Output	[0]	Control signal - Zero extend
Exc	Output	[0]	Unknown instruction

Halt	Output	[0]	HALT seen
FPInst	Output	[0]	Control signal - FP Instruction seen
FPIntCvtReg	Output	[1:0]	Control signal - FPIntCvtReg[0] - Write to Int Regfile. Source FP Reg. FPIntCvtReg[1] - Write to FP Regfile. Source Int Reg.
InstMemRead	Output	[0]	Control signal - LDCR seen
UnsignedOp	Output	[0]	Control signal - Unsigned operation (for unsigned compare and shift)
MemGran	Output	[0]	Control signal - Memory access granularity (00: Word, 01: Byte, 10: Half-word)
ctrl_err	Output	[0]	Non supported instruction seen
decode_err	Output	[0]	Tied to 0

2.2.3. Execute

Execute block contains the integer ALU. It outputs either the ALU computed output or an ALU error.

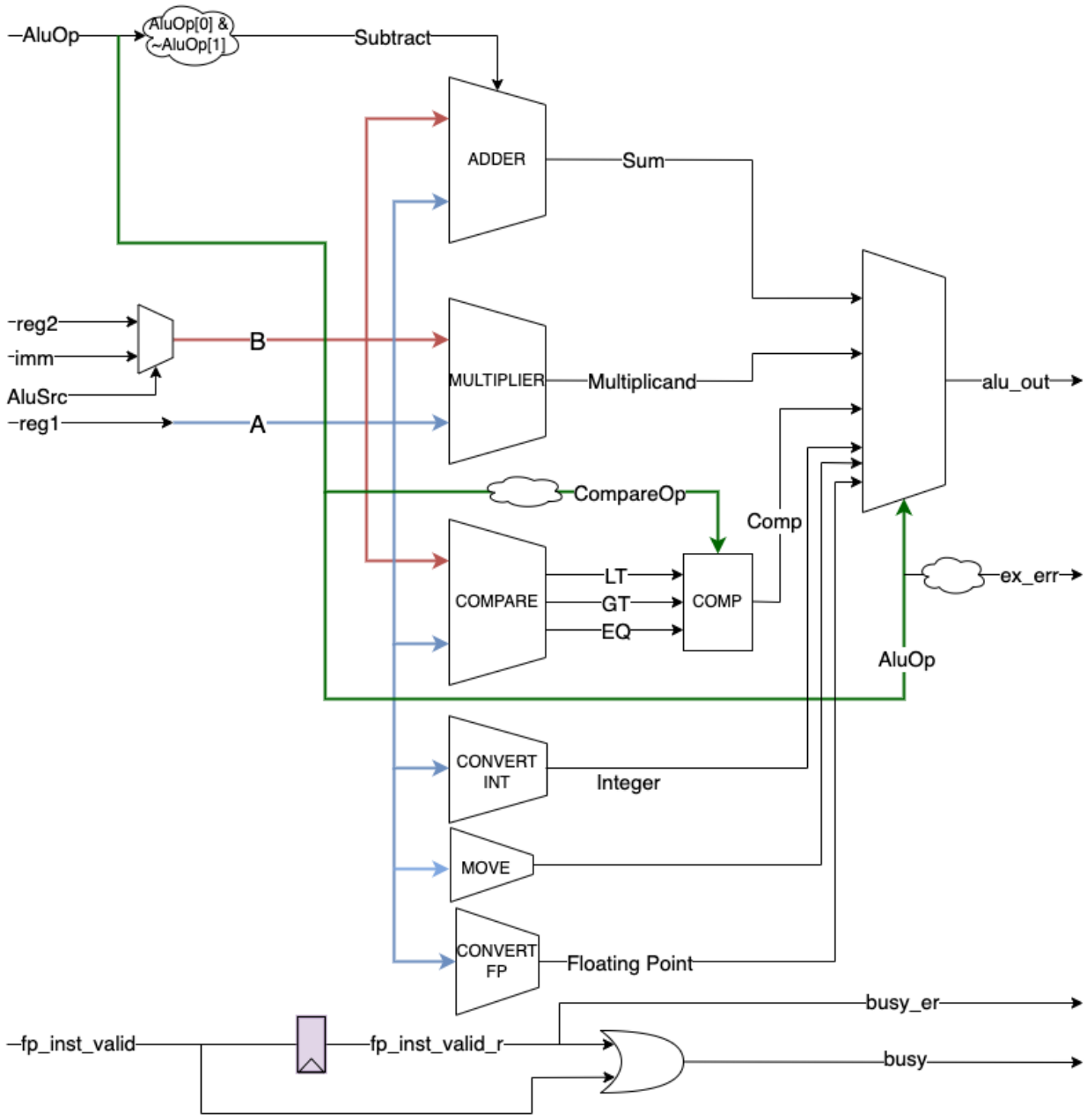
2.2.3.A Interface

Signal	**I/O**	**Width**	**Description**
reg1	Input	[31:0]	Read register source from decode
reg2	Input	[31:0]	Read register source from decode
imm	Input	[31:0]	Extended immediate from decode
pc_inc	Input	[31:0]	Next PC from IF. Use pc_inc as source for JType instructions.
alu_out	Output	[31:0]	ALU output
AluOp	Input	[4:0]	Operation to perform in Alu

JType	Input	[1:0]	JType instruction
AluSrc	Input	[0]	Immediate is a source
UnsingedOp	Input	[0]	Unsigned operation
ex_err	Output	[0]	Unsupported ALU operation

2.2.4. FP Execute

The FP execute block is a multi-cycle block which computes ALU operations for FP instructions.



2.2.4.A Registers

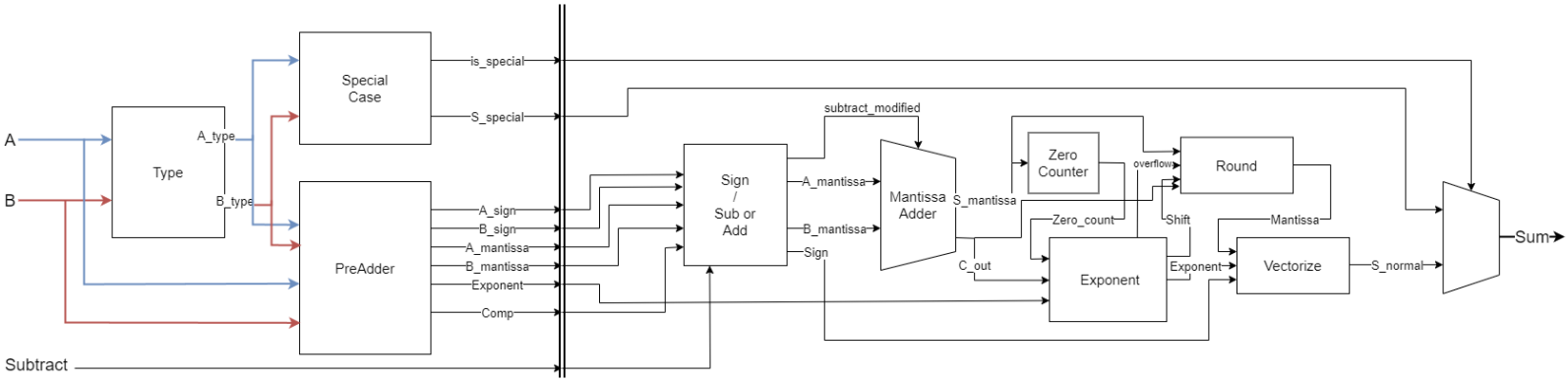
Register		Description
fp_inst_valid_r		Pipelined signal which determines if the FP EX is busy and early busy release.

2.2.4.B. Interface

Signal	I/O	Width	Description
reg1	Input	[31:0]	FP register source A
reg2	Input	[31:0]	FP register source B
Imm	Input	[31:0]	Immediate source value
alu_out	Output	[31:0]	Output
fp_inst_valid	Input	[0]	Is this instruction FP?
AluOp	Input	[3:0]	ALU operation
AluSrc	Input	[0]	Is immediate source
ex_err	Output	[0]	Unsupported ALU operation
busy	Output	[0]	FP execute is busy
busy_er	Output	[0]	Early busy release. FP execute will be free next cycle

2.2.4.1 FP Adder

Pipelined after the preadder.



2.2.4.1.A Registers

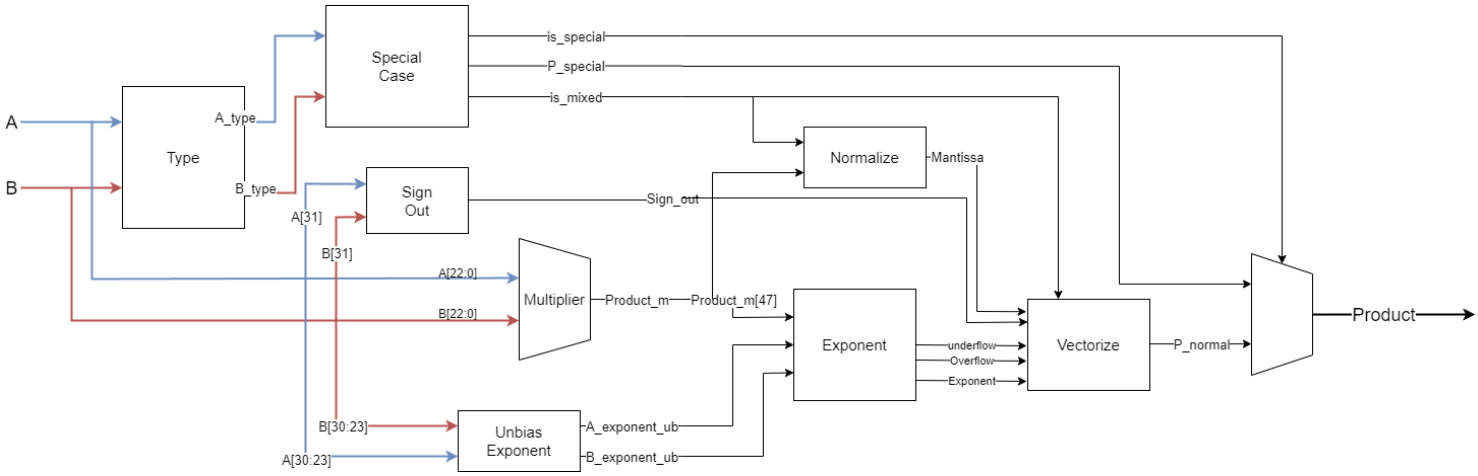
Register	Description
is_special	Pipelined signal which determines if the sum is a special case.

S_special	Pipelined signal which holds the sum when there is a special case.
A_sign	Pipelined signal which holds the sign of A.
B_sign	Pipelined signal which holds the sign of B.
A_mantissa	Pipelined signal which holds the mantissa of A
B_mantissa	Pipelined signal which holds the mantissa of B
Exponent	Pipelined signal which holds the exponent for the sum.
Comp	Pipelined signal which holds the comparison for A and B based on the scenario i.e. normal or subnormal
Subtract	Pipelined signal which determines whether it is addition or subtraction

2.2.4.1.B Interface

Signal	**I/O**	**Width**	**Description**
A	Input	[31:0]	FP register source A
B	Input	[31:0]	FP register source B
Sum	Output	[31:0]	Sum of A and B

2.2.4.2 FP Multiplier



2.2.4.2.B Interface

Signal	I/O	Width	Description
A	Input	[31:0]	FP register source A
B	Input	[31:0]	FP register source B
Product	Output	[31:0]	Product of A and B

2.2.5. Hazard

The hazard detection unit stalls the processor if there are register dependencies which can't be solved through forwarding. It also stalls the processor if a FP instruction is decoded and the FP execute unit is busy.

2.2.5.A. Interface

Signal	I/O	Width	Description
IF_ID_reg1	Input	[4:0]	Fetched instruction register A index
IF_ID_reg2	Input	[4:0]	Fetched instruction register B index
ID_EX_regw	Input	[4:0]	Decoded instruction register write index - out of decode stage
ID_FEX_regw	Input	[4:0]	Decoded instruction FP register write index - out of decode stage

ID_FEX2_regw	Input	[4:0]	Decoded instruction FP register write index - out of FEX1 stage
EX_MEM_regw	Input	[4:0]	Executed instruction register write index - out of execute stage
ID_EX_ctrl_regw	Input	[0]	Is decoded instruction writing a register - out of decode stage
ID_FEX_ctrl_RegWrite_out	Input	[0]	Is decoded instruction writing a FP register - out of decode stage
ID_FEX2_ctrl_RegWrite_out	Input	[0]	Is decoded instruction writing a FP register - out of FEX1 stage
EX_MEM_ctrl_regw	Input	[0]	Is executed instruction writing a register - out of execute stage
IF_ID_is_branch	Input	[0]	Branch instruction decoded - early indication
IF_ID_is_fp_store	Input	[0]	FP Store instruction decoded - early indication
IF_ID_is_fp_ex	Input	[0]	FP instruction going through FEX decoded - early indication
ID_EX_is_load	Input	[0]	Load instruction decoded
FEX_busy	Input	[0]	FP Execute is busy
FEX_busy_er	Input	[0]	FP Execute is busy - early release
ID_FEX_ctrl_FpInst	Input	[1]	FP Inst decoded in decode
ID_FEX_ctrl_FPIntCvtReg	Input	[1:0]	Which register file is written by FP instruction
ID_FEX2_ctrl_FpInst	Input	[1]	FP Inst in FEX1
ID_FEX2_ctrl_FPIntCvtReg	Input	[1:0]	Which register file is written by FP instruction

stall	Output	[0]	Stall when hazard detected

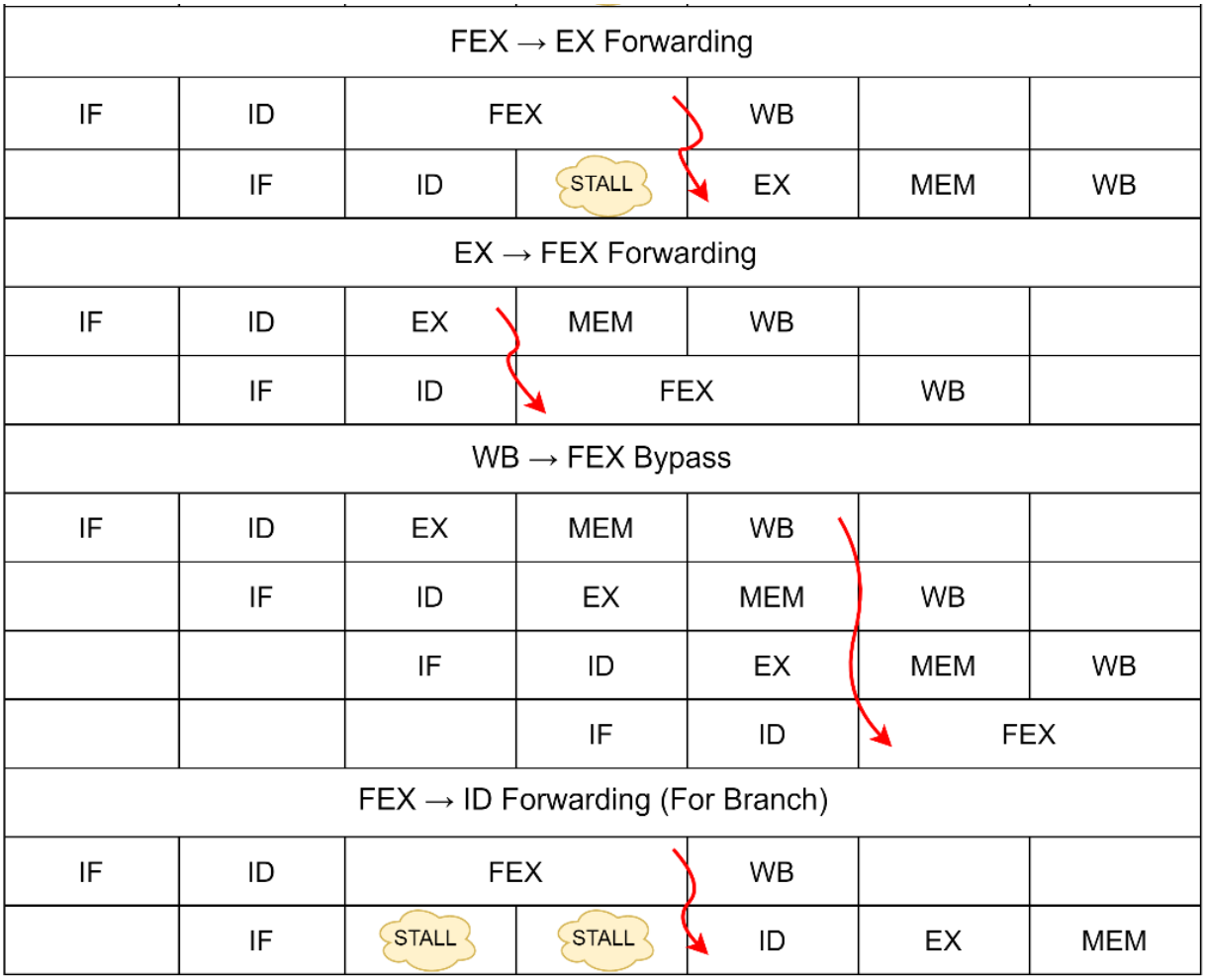
2.2.6. Forward

The forward block computes the forwarding and bypass signals for the integer and FP pipeline.

2.2.6.A Pipeline

The following pipeline diagrams show all the stall, forwarding and bypass cases WI-23 supports -

EX → EX Forwarding						
IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB	
MEM → EX Forwarding						
IF	ID	EX	MEM	WB		
	IF	ID	STALL	EX	WB	
EX → ID Forwarding (For Branch)						
IF	ID	EX	MEM	WB		
	IF	STALL	ID	EX	MEM	WB
WB → EX Bypass						
IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	
			IF	ID	EX	MEM
FEX → FEX Forwarding						
IF	ID	FEX		WB		
	IF	ID	STALL	FEX		WB
MEM → FEX Forwarding						
IF	ID	EX	MEM	WB		
	IF	ID	STALL	FEX		WB



2.2.6.B Interface

Signal	I/O	Width	Description
IF_ID_reg1	Input	[4:0]	Fetched instruction register A index
IF_ID_reg2	Input	[4:0]	Fetched instruction register B index
ID_EX_reg1	Input	[4:0]	Decoded instruction register A index
ID_EX_reg2	Input	[4:0]	Decoded instruction register B index
ID_FEX_reg1	Input	[4:0]	Decoded instruction FP register A index

ID_FEX_reg2	Input	[4:0]	Decoded instruction FP register B index
EX_MEM_regw	Input	[4:0]	Executed instruction register write index
MEM_WB_regw	Input	[4:0]	Instruction register write index out of MEM stage
FEX_WB_regw	Input	[4:0]	Instruction register write index out of FEX stage
EX_MEM_ctrl_regw	Input	[0]	Is register being written - out of EX stage
MEM_WB_ctrl_regw	Input	[0]	Is register being written - out of MEM stage
FEX_WB_ctrl_regw	Input	[0]	Is register being written - out of FEX2 stage
ID_EX_ctrl_MemWrite	Input	[0]	Store instruction decoded
ID_EX_ctrl_FpInst	Input	[0]	FP instruction decoded
ID_FEX_ctrl_FpInst	Input	[0]	FP instruction decoded
EX_MEM_ctrl_FpInst	Input	[0]	FP inst - EX to MEM
MEM_WB_ctrl_FpInst	Input	[0]	FP inst - MEM to WB
MEM_WB_ctrl_MemToReg	Input	[0]	Load instruction from MEM to WB
FEX_WB_ctrl_FpInst	Input	[0]	FP inst - FEX to WB
FEX_WB_ctrl_MemRead	Input	[0]	Load inst - FEX to WB
FEX_WB_ctrl_FPIntCvtReg	Input	[1:0]	Which register file is written from FP inst
ID_FEX_ctrl_FPIntCvtReg	Input	[1:0]	Which register file is written from FP inst

IF_ID_ctrl_FPIntCvtReg	Input	[1:0]	Which register file is written from FP inst
IF_ID_ctrl_FPInst	Input	[0]	Early indication of FP inst
frwrd_MEM_EX_opA	Output	[0]	Forward source A from EX -> EX stage
frwrd_MEM_EX_opB	Output	[0]	Forward source B from EX -> EX stage
frwrd_WB_EX_opA	Output	[0]	Forward source A from MEM -> EX stage
frwrd_WB_EX_opB	Output	[0]	Forward source B from MEM -> EX stage
frwrd_EX_ID_opA	Output	[0]	Forward source A from EX -> ID stage (for branch)
frwrd_FEX_ID_opA	Output	[0]	Forward source A from FEX -> ID stage (for branch)
bypass_reg1	Output	[0]	Bypass source A from WB -> EX
bypass_reg2	Output	[0]	Bypass source B from WB -> EX
fp_bypass_reg1	Output	[0]	Bypass source A from WB -> FEX
fp_bypass_reg2	Output	[0]	Bypass source B from WB -> FEX
fp_int_bypass_reg1	Output	[0]	Bypass source A from WB -> FEX (FP)
fp_int_bypass_reg2	Output	[0]	Bypass source B from WB -> FEX (FP)
frwrd_MEM_FEX_opA	Input	[0]	Forward source A from EX -> FEX
frwrd_MEM_FEX_opB	Input	[0]	Forward source B from EX -> FEX
frwd_WB_FEX_opA	Output	[0]	Forward source A from FEX2 -> FEX1 stage or MEM -> FEX1
frwd_WB_FEX_opB	Output	[0]	Forward source B from FEX2 -> FEX1 stage or MEM -> FEX1

frwd_int_WB_EX_opA	Output	[0]	Forward source A from MEM -> FEX1 stage
frwd_int_WB_EX_opB	Output	[0]	Forward source B from MEM -> FEX1 stage
frwd_fp_WB_EX_opA	Output	[0]	Forward source A from FEX2 -> EX stage
frwd_fp_WB_EX_opB	Output	[0]	Forward source B from FEX2 -> EX stage

2.2.7. MEM

This top-level logic assigns the output data memory signals from CPU to WI23 top.

2.2.8. Write Back

This top-level logic assigns the ‘write_in’ and ‘fp_write_in’ signals to the decode block.

2.3. RST

Reset Synchronizer. Double-FF synchronizer synchronizes the asynchronous reset from the FPGA.

2.3.A. Registers

Register	**Width**	**Description**
RST_n_ff1	[0]	First FF of the synchronizer

2.3.A. Interface

Signal	**Input/Output**	**Width**	**Description**
clk	Input	[0]	Clock

RST_n	Input	[0]	Asynchronous reset from the FPGA.
rst_n	Output	[0]	Synchronized reset.

2.4. IMEM

True dual-port 32kB Instruction memory.

- Port 1: Instruction read
- Port 2: Data read for LDCR

2.4.A. Registers

Register	**Width**	**Description**
mem_r [8192]	[31:0]	Instruction Memory
inst_r	[31:0]	Instruction Read
data_r	[31:0]	Data Read

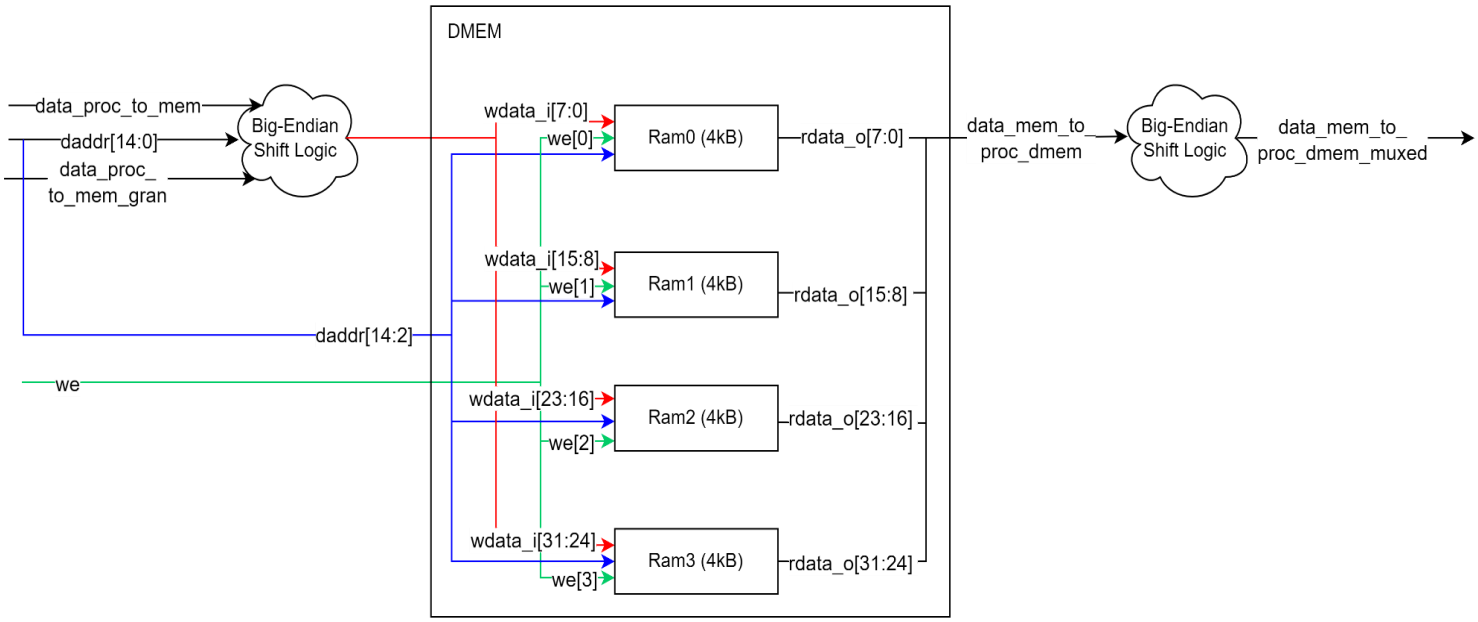
2.4.B. Interface

Signal	**Input/Output**	**Width**	**Description**
clk	Input	[0]	clk
addr_i	Input	[12:0]	PC.
inst_o	Output	[31:0]	Instruction read from the instruction memory.
daddr_i	Input	[12:0]	Address to read from IM for LDCR
data_o	Output	[31:0]	Data read from IM for LDCR

2.5. DMEM

Big-endian 16kB memory.

- Four 4kB (4k entries x 1 byte) memory banks to support sub-word accesses.



2.5.A. Registers

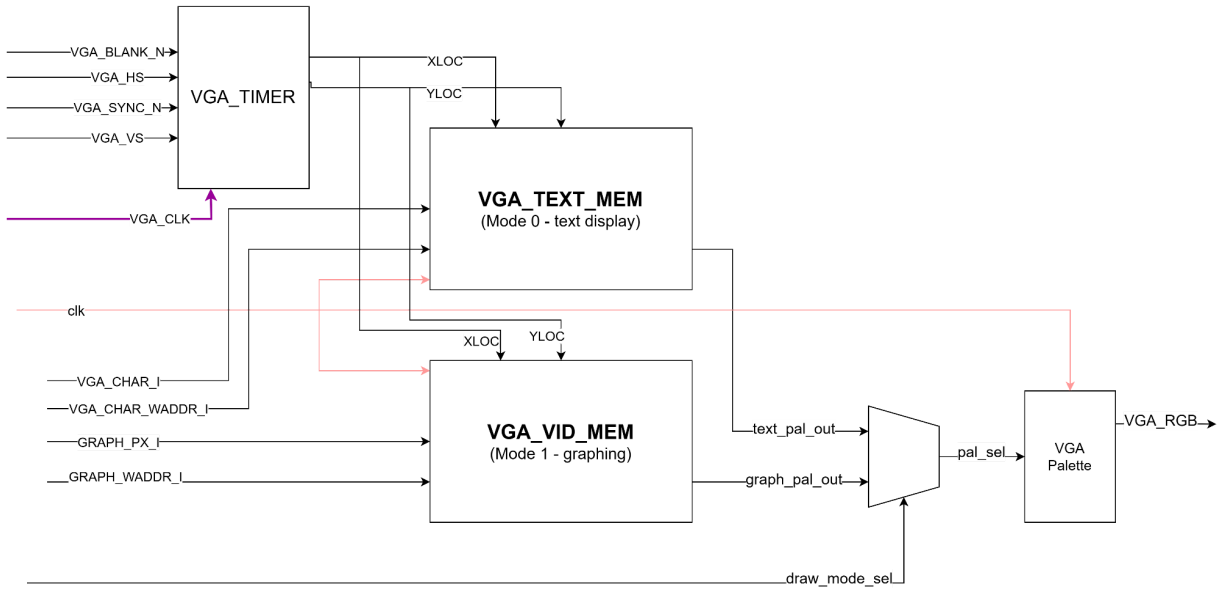
Register	Width	Description
Ram0: mem_r [4096] Ram1: mem_r [4096] Ram2: mem_r [4096] Ram3: mem_r [4096]	[7:0]	Data Memory

Ram0: rdata_r Ram1: rdata_r Ram2: rdata_r Ram3: rdata_r	[7:0]	Data Read

2.5.B. Interface

Signal	**Input/Output**	**Width**	**Description**
clk	Input	[0]	Clock
addr_i	Input	[13:0]	Address to read from data memory.
we_i	Input	[3:0]	Qualified write enable to data memory. Only asserted if not writing to a memory mapped peripheral.
wdata_i	Input	[31:0]	Write data to data memory.
rdata_o	Output	[31:0]	Read data from data memory.

2.6. VGA GPU



The VGA GPU / driver is designed to both offer practical drawing options for a graphing calculator as well as extremely minimal framebuffer memory usage:

- 80x30 VGA text mode. The character set is stored in a ROM set (by default, the classic IBM 8x16 font is used). Uses 16 bits per character, resulting in a 4.8kb buffer.
    - 8 bits of text memory form the low bits, and 8 bits of palette data (3 for the BG colors, 4 for the FG colors, 1 for cursor select [reserved, but goes unused]).
- Quarter-size line-drawing framebuffer (320x240) for visualizing equations. Each pixel has 4 bits of palette info, resulting in a 38.4kb buffer.
    - Albeit, each palette nybble is actually mapped to a full byte, resulting in twice the actual size.

VGA draw modes are changed via a line in, reflecting a memory mapped address value.

A notable bug / codec issue of the VGA text mode unit is that there appears to be aliasing that results in garbage pixels being drawn. This is strangely not a universal issue, and is not present on some boards.

Another bug is the VGA text mode having a “shadow” color, drawing an additional color next to pixels. The shadow color itself seems to be random, and it is unknown why this happens.

2.6.A. Registers

Reserved address spaces for VGA_VMEM_TEXT and VGA_VMEM_GRAPH are tentative.

Register	Address (range)	Description
VGA_TEXT_BUFFER	0xFFFEC140-0xFFFED3FF	VGA Text Mode framebuffer. Write-only.
VGA_GRAPH_BUFFER	0xFFFED400-0xFFFFFFFF	VGA Graph Mode framebuffer. Write-only.
VGA_CONFIG	0xFFFEC134	Sets the VGA mode line = bit 0. Write-only.

2.6.B. Interface

Signal	Input/Output	Width	Description
VGA_CLK	input	[0]	Timing for the VGA unit.
VGA_BLANK_N	output	[0]	Asserted low between frames when nothing is being drawn.
VGA_HS	output	[0]	Active low horizontal sync for positioning the video signal.
VGA_SYNC_N	output	[0]	Controls V-sync.
VGA_VS	output	[0]	Active low vertical sync for positioning the video signal.
VGA_RGB	output	[23:0]	Red, green, and blue components for the VGA video signal.
draw_mode_sel_i	input	[0]	Selects between Mode 0 (text mode) and Mode 1 (graph mode).
VGA_char_i	input	[15:0]	VGA character data in (contains char + color data) to draw to the graph
VGA_char_waddr_i	input	[11:0]	Address to write to the text framebuffer
graph_px_i	input	[3:0]	Palette data in to draw to the graph
graph_waddr_i	input	[18:0]	Address to write to the framebuffer

2.7. PS/2 Keyboard

Interface to the PS/2 keyboard I/O. Despite plans for interrupts to be implemented, the PS/2 keyboard is entirely polled, and consumes input upon a programmer reading in the MMIO address.

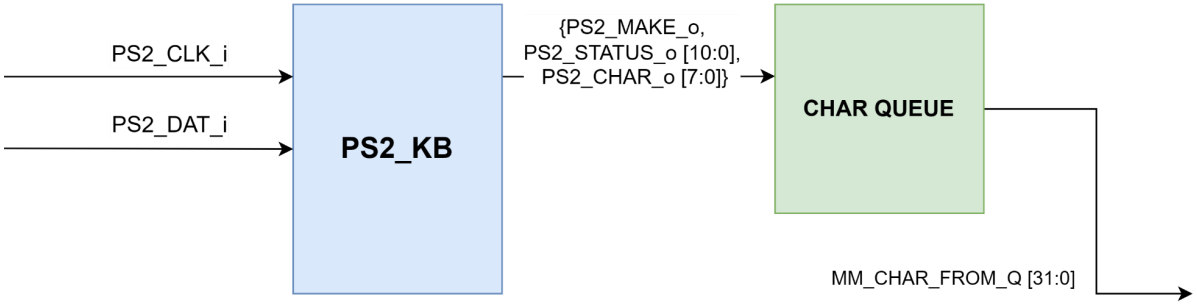
2.7.A. Registers

Register	Address (range)	Description
PS2_STATUS	0xFFFEC138	Deprecated. Was going to read the status codes, but since these are bundled with PS2_CHAR, it returns the same value.
PS2_CHAR	0xFFFEC13C	Read-only MMAP register, used by the programmer to get the last key pressed, along with the status codes. Notably, upon a read, the entry in the buffer **is consumed and will return the next value (or 0) on the next read**. The 32-bit result returned is laid out in this way: 31 ..0 {11'b0, make, status[10:0], char[7:0]}

2.7.B. Interface

The PS/2 keyboard driver is directly connected to a queue in the top level that has info about keystrokes pushed onto it, and popped off via a programmer reading the MMAP'd register.

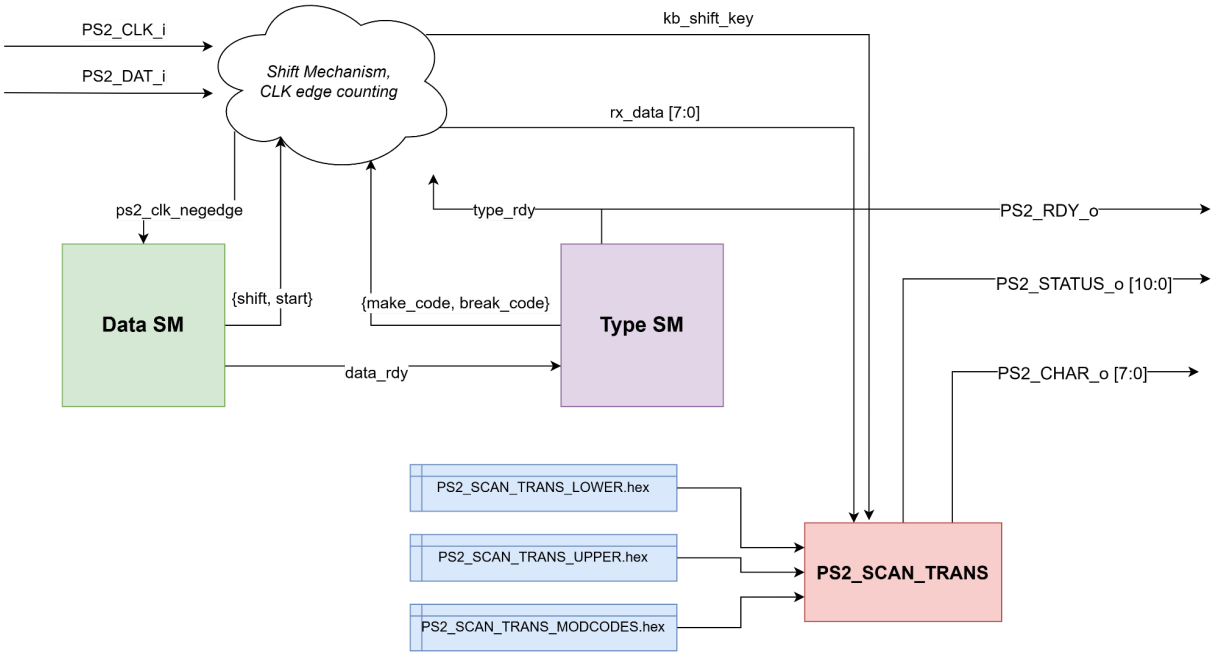
-TOP LEVEL-



Top-Level Look

Signal	Input/Output	Width	Description
PS2_DAT	input	[0]	Data from the PS/2 keyboard serial interface.
PS2_CLK	input	[0]	Clock from the PS/2 keyboard serial interface.
PS2_STATUS_o	input (queue), output (to MMAP)	[10:0]	Status codes read by PS2_SCAN_TRANS.
PS2_CHAR_o	input (queue), output (to MMAP)i	[7:0]	Character code translated by PS2_SCAN_TRANS.
PS2_MAKE_o	input (queue), output (to MMAP)	[0]	Outputs 1 if make (pressed), 0 if break (released).
MM_CHAR_FROM_Q	output	[31:0]	Output of the queue.

Internal View - PS/2 Peripheral



2.8. Timer

Simple peripheral that counts VBLANK posedges. While simple, it is a deceptively useful peripheral to be polled by the programmer, allowing simple QOL features such as animations per-frame, and complex ones such as entire games running at a constant 60 FPS.

2.7.A. Registers

Register	Address (range)	Description
TIMER	0xFFFEC130	Number of VBLANK cycles since power-on.

2.9. Misc logic

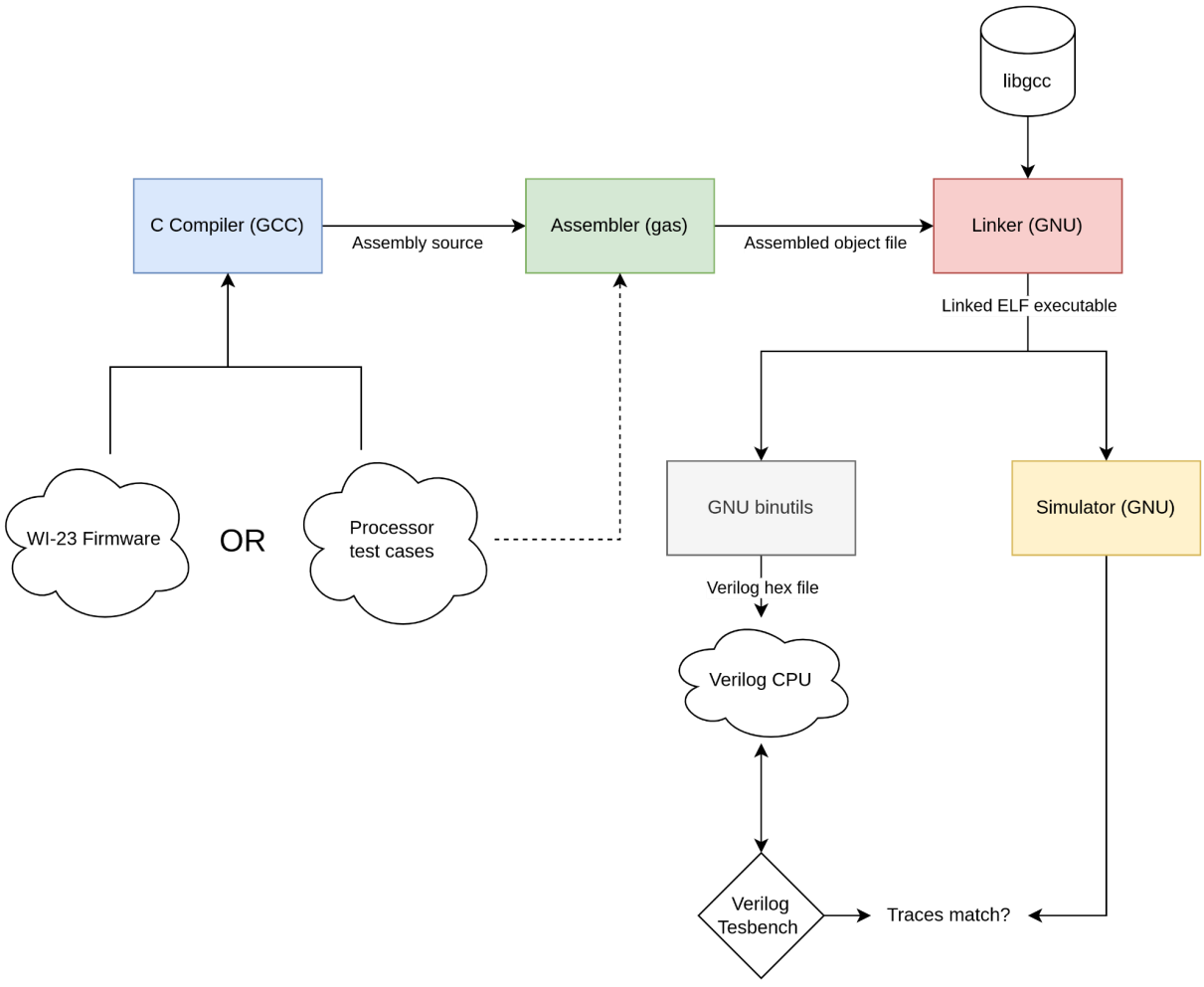
1. Drive I/O Address to memory mapped peripherals (SPART, VGA and PS/2)
2. Drive read and write enable signals to the data memory and memory mapped peripherals. This is dependent on the address being accessed.
3. Drive data read from data memory, SPART, VGA and PS/2 to the CPU
4. Coalesce IRQ inputs from all the memory mapped peripherals
5. Drive other I/O interface signals.

3. Software

This section will discuss the software employed in the WI-23, both on a host machine and on the board itself.

3.1 WI-23 Toolchain

Here, the toolchain refers to all of the software written to support developing programs for WI-23. All of this software runs on a host PC, not on the WI-23 itself. The diagram below outlines a comprehensive flow from C code to machine code using this toolchain, specifically from the context of verification.



Below is an explanation of each of the components in the diagram.

1. **WI-23 Firmware**: Firmware that runs on our processor on the DE1 board, providing the calculator functionality
2. **Test cases**: Unit tests (written by us) that can be run through the toolchain to facilitate verification. These can come in the form of self-contained C code or assembly files.

3. **Compiler**: WI23 C compiler powered by a GCC backend. Supports optimized code by virtue of GCC's rich optimization passes at the intermediate level representation (before our code.)
4. **Assembler**: Assembler for WI-23 using the GNU as infrastructure.
5. **Linker**: GNU linker for WI-23. Allows for the linking of multiple compiled C files together.
6. **libgcc:** Runtime support library for missing instructions (integer & float divide, other soft FP functions.) Also contains the startup routine crt0.S (handwritten), which sets the initial stack and frame pointers, and copies the data segment from instruction memory to data memory. This allows for the compiled C code to treat the address spaces as unified, when loading from IMEM would normally require using LDCRs. This is done because the compiler can't determine address spaces automatically. Finally, crt0.S jumps to the main function.
7. **Binutils**: binutils is a wide range of utilities for executable formats, the main utility used here is objcopy which can take an ELF and generate a loadable verilog hex file.
8. **Simulator:** simulator for the ISA using GNU sim infrastructure. Takes in an ELF executable as input and generates traces based on instructions run.
9. **Verilog CPU:** self explanatory
10. **Verilog testbench:** A testbench for the Verilog CPU design. This does not interface with the top-level SoC model with the other peripherals instantiated, only the CPU itself. It will generate traces in the same format as the GNU simulator so that they can be compared.

Note on standard library presence: libgcc is **NOT** the standard C library! There were originally plans to support newlib, a minimal standard library intended for embedded applications. While it did compile successfully at one point, it was scrapped due to being deemed mostly unnecessary and complicating the development testing process: if you are planning to develop one application only for your system, why not move all functionality into that application itself? In the end, we ended up simply recreating any functionality we would have needed from the standard library in our application itself.

3.1.1 Compiler

As mentioned in the toolchain section above, the compiler uses GCC. Specifically, it uses GCC's well defined interface for creating backends, which enables it to reuse all of the existing GCC codebase for parsing C and optimizing code at the intermediate representation level. A more in-depth description of the roles that the backend files contributed as part of the project play can be found in the README of the toolchain repository itself: https://github.com/panther03/wi23-gcc/tree/master#readme. Essentially, these backend files use a combination of C++ code and some of GCC's own custom "machine description" language to tell the rest of the compiler how to transform instructions in the intermediate representation to WI-23 assembly instructions. Then, as the toolchain diagram shows, the compiler spits out an assembly file, and this goes through the assembler and linker to produce a full ELF executable file.

3.1.2 Simulator

Our WI-32 simulator accurately replicates the functionality of the real hardware to test the compiler and firmware, and to validate the hardware. The software tool prints out information about each instruction, and the register or memory addresses that it changes, so that the trace can be inspected or compared with a trace from Modelsim.

For every instruction, the simulator classifies it by instruction type, and parses any source registers, destination registers, and immediate. Then, the simulator preforms the requested operation to a data register, floating point register, or memory location, and updates the program counter. This continues until the simulator reaches a halt instruction.

Expected outputs from the simulator and SV TB:

```
insn:    PC: @0x0000 Inst: ADD RegWrite: R1 RegValue: 0xaaaaaaaa LoadAddr:
0xxxxxxxxx LoadValue: 0xxxxxxxxx StoreAddr: 0xxxxxxxxx StoreValue: 0xxxxxxxxx

insn:    PC: @0x0004 Inst: LD RegWrite: R2 RegValue: 0xbbbbbbbb LoadAddr:
0x00000000 LoadValue: 0xbbbbbbbb StoreAddr: 0xxxxxxxxx StoreValue: 0xxxxxxxxx

insn:    PC: @0x0008 Inst: BEQZ RegWrite: XX RegValue: 0xxxxxxxxx LoadAddr:
0xxxxxxxxx LoadValue: 0xxxxxxxxx StoreAddr: 0xxxxxxxxx StoreValue: 0xxxxxxxxx
```

Pass Case:

```
1
2 ---------- Simulation trace ended ----------
3 PASSED! Simulation trace the same!
```

Fail Case:

```
1 --- wi23_tb_trace.log
2 +++ wi23_sim_trace.log
3 @@ -31,2 +31,2 @@
4 -insn:     PC: @0x007c Inst: FMUL RegWrite: F23 RegValue: 0xffff0000 LoadAddr: 0xxxxxxxxx LoadValue: 0xxxxxxxxx StoreAddr: 0xxxxxxxxx StoreValue: 0xxxxxxxxx
5
6 -insn:     PC: @0x0080 Inst: IMOVF RegWrite: R4 RegValue: 0xffff0000 LoadAddr: 0xxxxxxxxx LoadValue: 0xxxxxxxxx StoreAddr: 0xxxxxxxxx StoreValue: 0xxxxxxxxx
7
8 +insn:     PC: @0x007c Inst: FMUL RegWrite: F23 RegValue: 0xffffccdd LoadAddr: 0xxxxxxxxx LoadValue: 0xxxxxxxxx StoreAddr: 0xxxxxxxxx StoreValue: 0xxxxxxxxx
9
10 +insn:     PC: @0x0080 Inst: IMOVF RegWrite: R4 RegValue: 0xffffccdd LoadAddr: 0xxxxxxxxx LoadValue: 0xxxxxxxxx StoreAddr: 0xxxxxxxxx StoreValue: 0xxxxxxxxx
11
12
13 ---------- Simulation trace ended ----------
14 FAILED! Simulation trace difference detected!
```

4. Firmware

4.1 Front End Firmware

The front-end firmware takes a simple state-based approach to drawing each calculator function mode and taking in input. The VGA text mode is used for text-based menus, while graph mode is switched to when an equation is successfully graphed in “Graph” mode. This is done by writing to a memory-mapped register.

Within a window state, input is taken in or removed character by character into a struct containing a buffer depending on the memory-mapped keyboard register. ASCII data is saved as the lower 8 bits, while the status bits and the MAKE line are saved as the upper 10 and the 19th bit respectively.

With the “Calculate” mode, the user is able to input a full line of characters to be tokenized and parsed as an equation (see 4.3). Pressing ENTER will evaluate the equation, showing the result in green text, or printing “SYNTAX ERROR” in red text if there was an error.

WIth the “Equation” mode, up to four equations Y1-Y4 may be input, which will be graphed with the “Graph” mode. (see 4.4). If an error arises, a special error window will be displayed, telling the user where the error occurred.

In addition, “Settings” mode allows the user to enter in graph bounds to be used in “Graph” mode, and also an arbitrary value for X to be used in “Calculate” mode.

For animations, the number of VBLANK clock cycles that have passed since reset are saved and readable as a memory-mapped register. Animating frame-by-frame is as simple as reading in this register and either applying a bitmask or using modulo on the result. This is used for quality-of-life features like the blinking carat in “Calculate” and the spinning cursors in the “Equation” and “Settings” modes.

4.1.1 Tetris

Going one step further, it is possible to implement a basic game event loop by reading in VBLANK as often as possible, and executing one frame of game engine logic when the last value read doesn’t equal the current value. This is the basic principle that allows “Tetris” to be implemented as a calculator mode.

The “Tetris” implementation provided is a port written from the ground up. There are a few notable quirks of this version in particular:

- Mostly based on the implementation of Tetris released for the Nintendo Entertainment System.
    - Level drop speed is nearly the same table as the NES version, and stops progressing at Level 19.
    - Uses the Right-Handed Nintendo Rotation System.
    - No wall kicks, pieces snap instantly once they can’t move a cell down.
- Key bindings are mapped to ASD and JK for movement and rotation respectively instead of the standard Arrow Keys and SPACE.
    - Soft-drop behavior uses set-reset behavior: however, the soft drop flag isn’t cleared upon game reset, so if the player got a game over while holding the soft drop key, the next piece will (annoyingly) soft-drop down the well regardless of the player’s inputs.
- DAS (delayed auto shift) is provided via the typematic capabilities of the PS/2 keyboard itself.
    - The DAS from the keyboard is noticeably very slow, making high-speed play difficult.
    - Theoretically, DAS may be “charged” (having typematic mode activated by holding a directional key down long enough) during line clear and ARE.
- RNG for generating pieces is not based on a LFSR approach, but instead uses the current VBLANK count read as an 8-bit word modulo 7.

    - Theoretically, the RNG regenerates when a piece matches the last piece generated, but this does nothing as it calls the same *TIMER value.
- Pieces are drawn using VGA text mode.
- Lines are cleared with a slight, nearly unnoticeable gradient animation per text character, and are cleared one at a time. This takes 20 frames.
- Line delay (ARE) is 10 frames. This is (unintentionally) applied when you clear a line as well, making the time between one line clear (a single) and the next piece dropping 40 frames.

4.2 Parser Firmware

The parser utilizes the Shunting Yard Algorithm to convert an input into a queue of tokens in Reverse Polish Notation. First, an input is passed from the front end as a string. The string is parsed character by character to put each token into a queue in infix notation. Next, the input queue is passed through an implementation of the Shunting Yard algorithm to format it in the CPU/FPU-readable representation in Reverse Polish notation.

The parser supports addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^), as well as integers and floating point numbers in scientific notation and standard notation, and variables. The standard order of operations is strictly followed, and parentheses are also taken into account when converting the input into Reverse Polish Notation.

There are notable restrictions on the syntax accepted by the parser: Negative constants are allowed (as the fp_to_str routine supports them), but negative variables aren't, as no unary subtraction operator is supported (so while "-10.0" is OK, "-x" isn't, but "0-x" is completely fine, as well as "-1.0*x"). Implicit multiplication is also not supported, so typing "10x" will result in an error (but "10*x" is completely fine).

4.3 Equation Solving Firmware

Once equations are converted to Reverse Polish Notation, the firmware can solve them to produce the final result. The equation solver takes a queue storing the equation, as well as an optional floating point value as input. The floating point value may be used to replace any variables in the equation when generating a graph.

4.4 Graphing Firmware

The graphing firmware makes use of a graph function that gets the bounds for the graph, the color of the line, and the equation of the line to graph. It then writes out to the frame buffer for the vga display. First it draws the axes of the graph, then the line. It plots one pixel per column and connects them using Bresenham's line drawing algorithm. If the output for a given x is not within the bounds of the graph, it is simply not plotted. The graph will throw errors for bounds that don't make sense (i.e. y_mininum >

y_maximum) or if it is given a bad equation. The calculator is capable of graphing 4 lines. It also lets the user specify the bounds of the graph.

4.5 Data Structures and String Operations

Since the C compiler does not support the standard library, several data structures and string manipulation tools were created from scratch for use in the firmware. These tools include stacks, queues, and functions to convert floating points to and from strings.

4.5.1 String to Floating Point

Parser utilizes a string to floating point function that takes a string and outputs a floating point value and an error code if there was an issue with parsing the string. The strings can be just an integer (no decimal i.e. “10”), floating point (have a decimal point i.e. “13.5”), or scientific notation (i.e. “3.45e6”). A number can also be prefixed with a negative sign.

4.5.2 Floating Point to String

In outputting a floating point value to the screen, the float needs to be converted to a string. This is done with a function that takes a floating point value and outputs a string. If the floating point value is less than 1e7 and greater than 1e-5, it outputs the floating point values with 6 possible digits to the left of the decimal point and 5 possible values to the right of the decimal. If the floating point value is outside this range, it outputs a string with scientific notation.

5. Engineering Standards Employed in your Design

This table shows the engineering standards we have used in our design and the relevant components which correspond to the standards.

Standard	**Component**
IEEE 754 Floating Point (32 bit)	FP hardware
IBM VGA Text Mode, IBM 8x16 Fontset	VGA hardware module.
Infix Notation, Postfix Notation (Reverse Polish Notation)	Equation Parser/Solver

ISO/ANSI C standards	C Compiler

6. Potential Societal Impacts of Your Design

For years, the industry for graphing calculators in middle schools, high schools, universities, and much of the industry has been dominated by Texas Instruments and a few other large companies. In reality, the processor and software behind a graphing calculator are relatively simple and do not need to be so exclusive. Our WI-23 graphing calculator proves that if a small group of university students can develop a graphing calculator almost from scratch, basic and affordable graphing calculators for students should be more widely available.

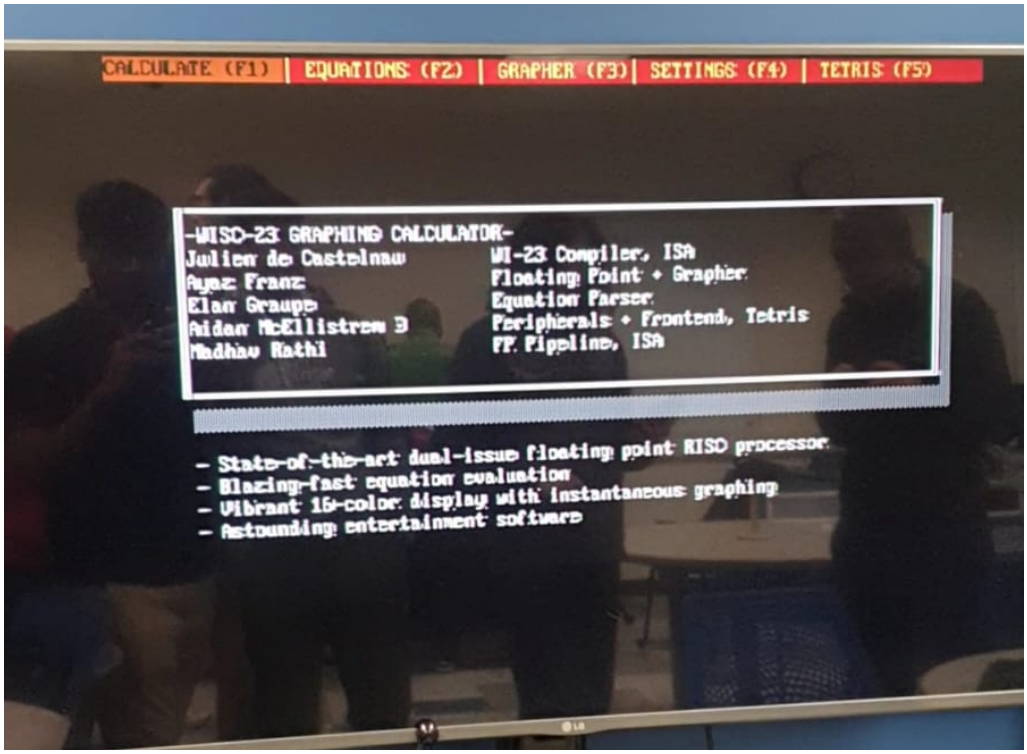
7. Final Application Demonstration

The calculator supports the following mathematical operations:

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Exponents

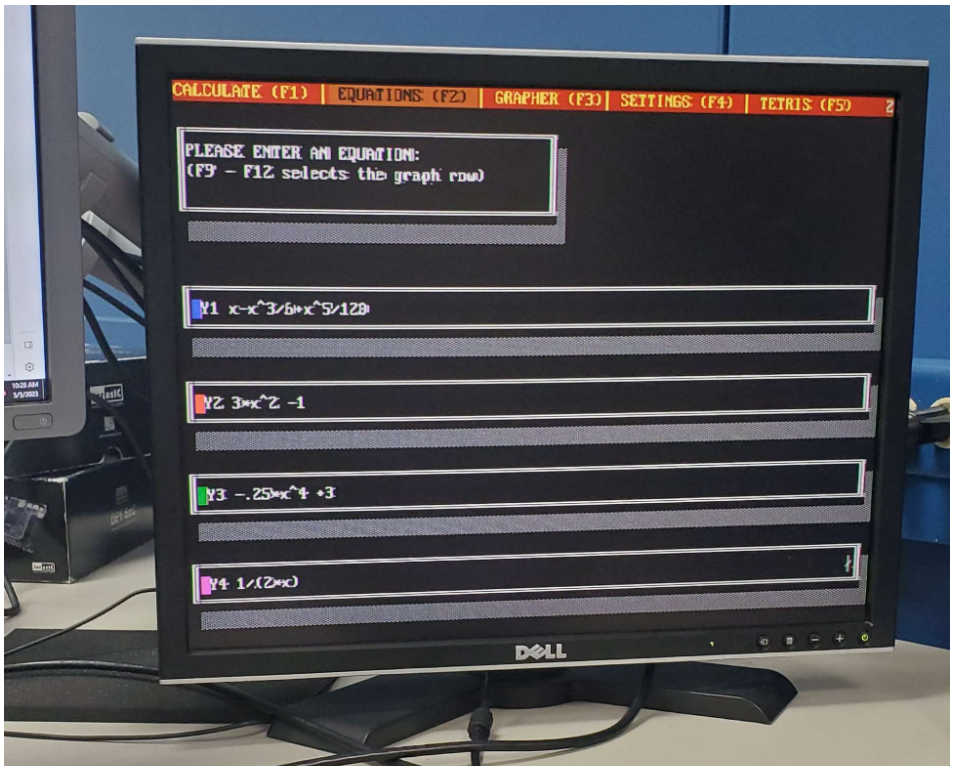
Calculate/Start tab:

Enter an equation to be solved.



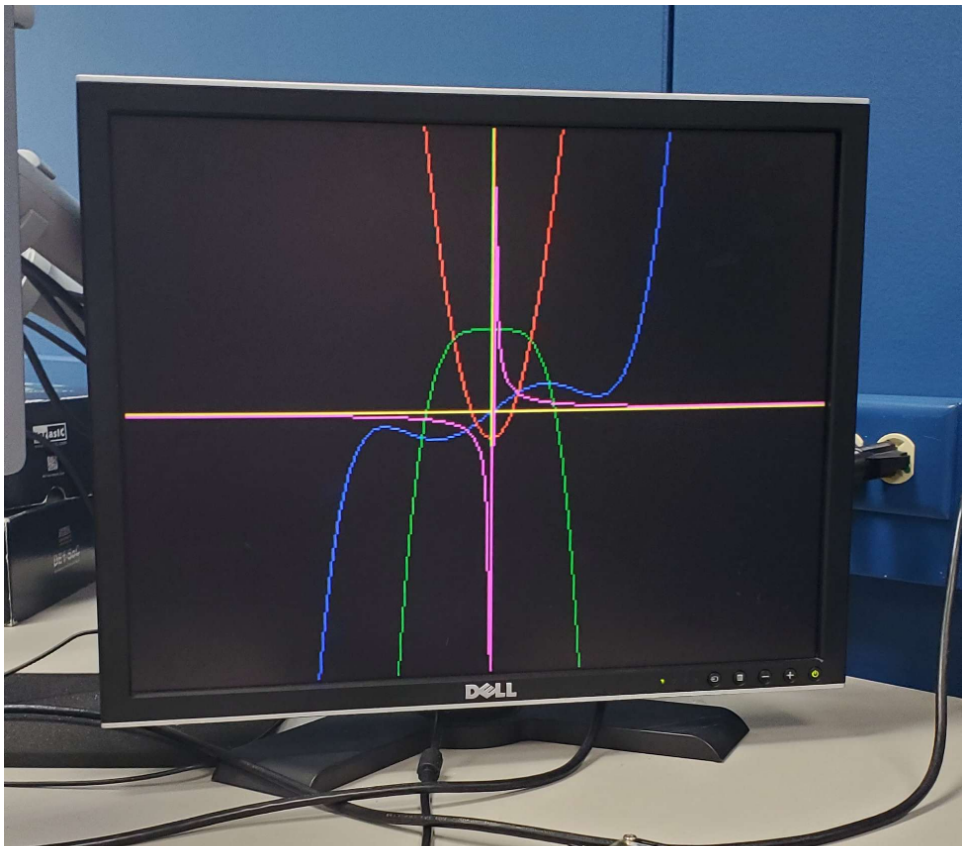
Equation tab:

Allows the user to write 4 equations which need to be graphed. Equation lines are chosen by F9-F12



Graph Tab:

Graphs the equations entered in the equation tab.

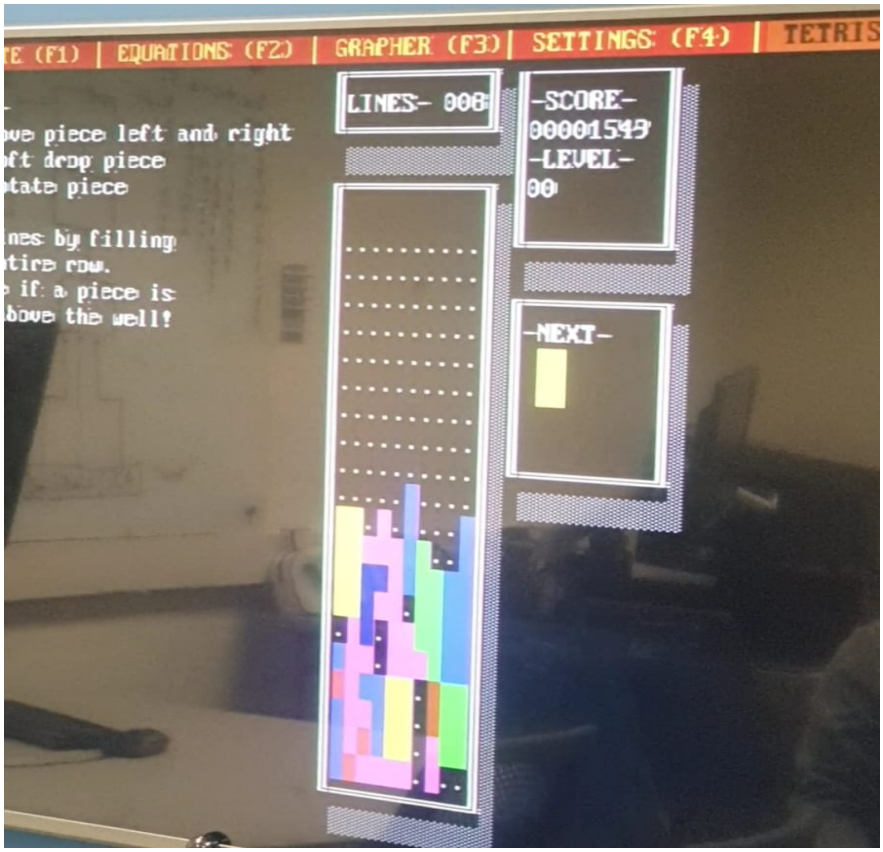


Settings Tab:

Changes the graph bounds.

Tetris Tab:

Play tetris and enjoy.


E (F1) | EQUATIONS (F2) | GRAPHER (F3) | SETTINGS (F4) | TETRIS
ve piece left and right
ft drop piece
tate piece
nes by filling
tire row.
if a piece is
bove the well!
LINES- 008
-SCORE-
00001543
-LEVEL-
00
-NEXT-

8. Contributions of Individuals

Team Member	Contributions
Julien de Castelnau	GNU compiler, assembler, linker, simulator backend. Various automation & build scripts. ISA & compiler/assembler implementation.
Ayaz Franz	FP execute. Grapher module. FP/string conversion modules
Elan Graupe	Equation parser. Equation solver. Simulator.
Aidan McEllistrem	VGA module. PS/2 module. Top-level memory map. Corresponding software drivers, software front-end, tetris C implementation.
Madhav Rathi	CPU pipeline w/ verification. ISA & hardware implementation. Team lead managing JIRA & project timelines.