The WI-23 General Purpose Graphing Calculator

Wisconsin Instruments

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

Project Interface Document

Table of Contents

1. 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.

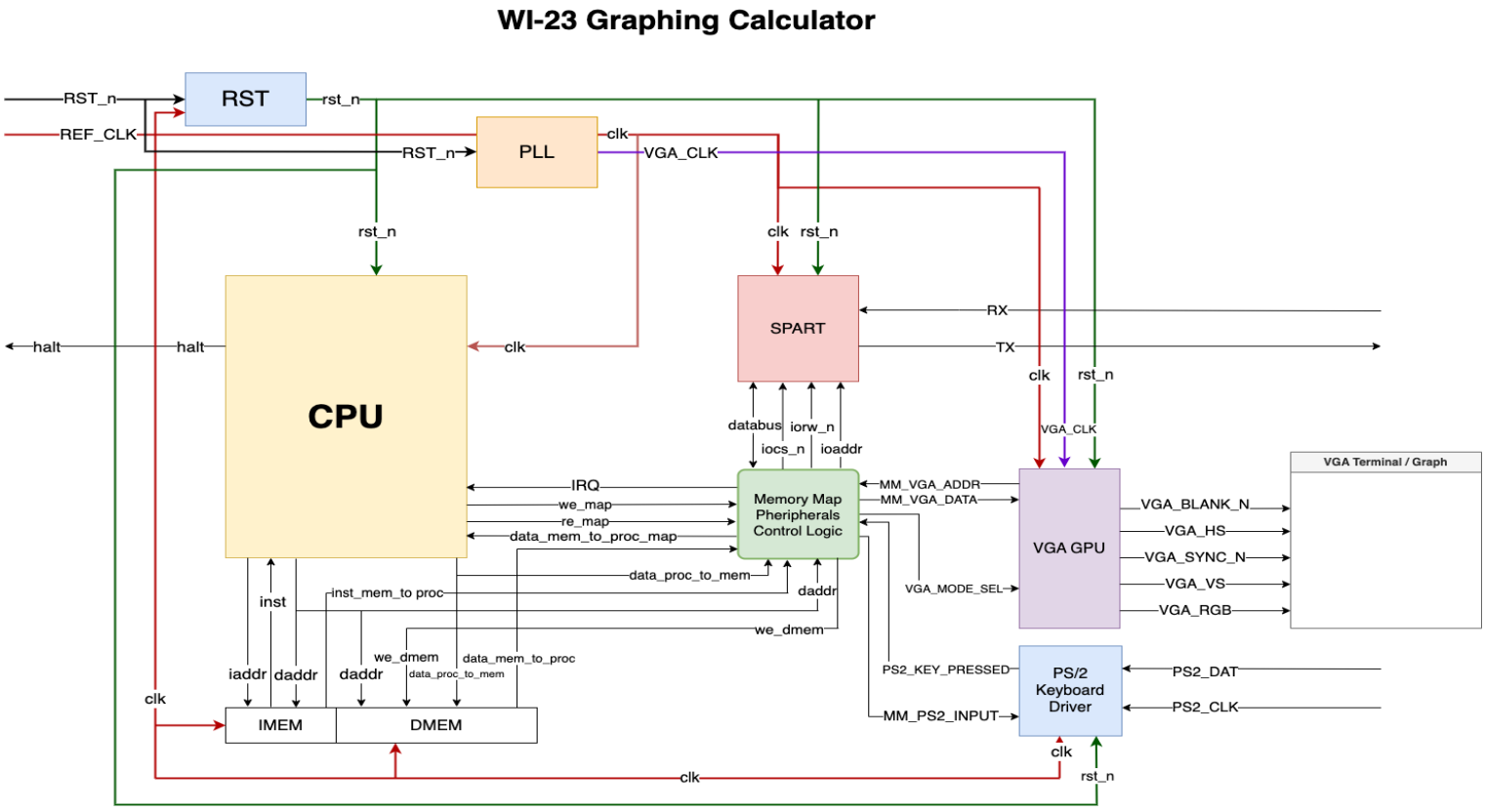
A wide range of floating-point math operations are provided for use by the graphing calculator REPL software (and supported by the underlying hardware), and will be graphed in an equation editor onto a VGA display.

The top-level module contains all the hardware components for WI-23 to work.

1.1 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. SPART: UART interface to help with debug. Specific debug usage is TBD
6. Memory Map Peripheral Logic: Logic to collate and drive all memory map interface signals.



1.1.A. Interface

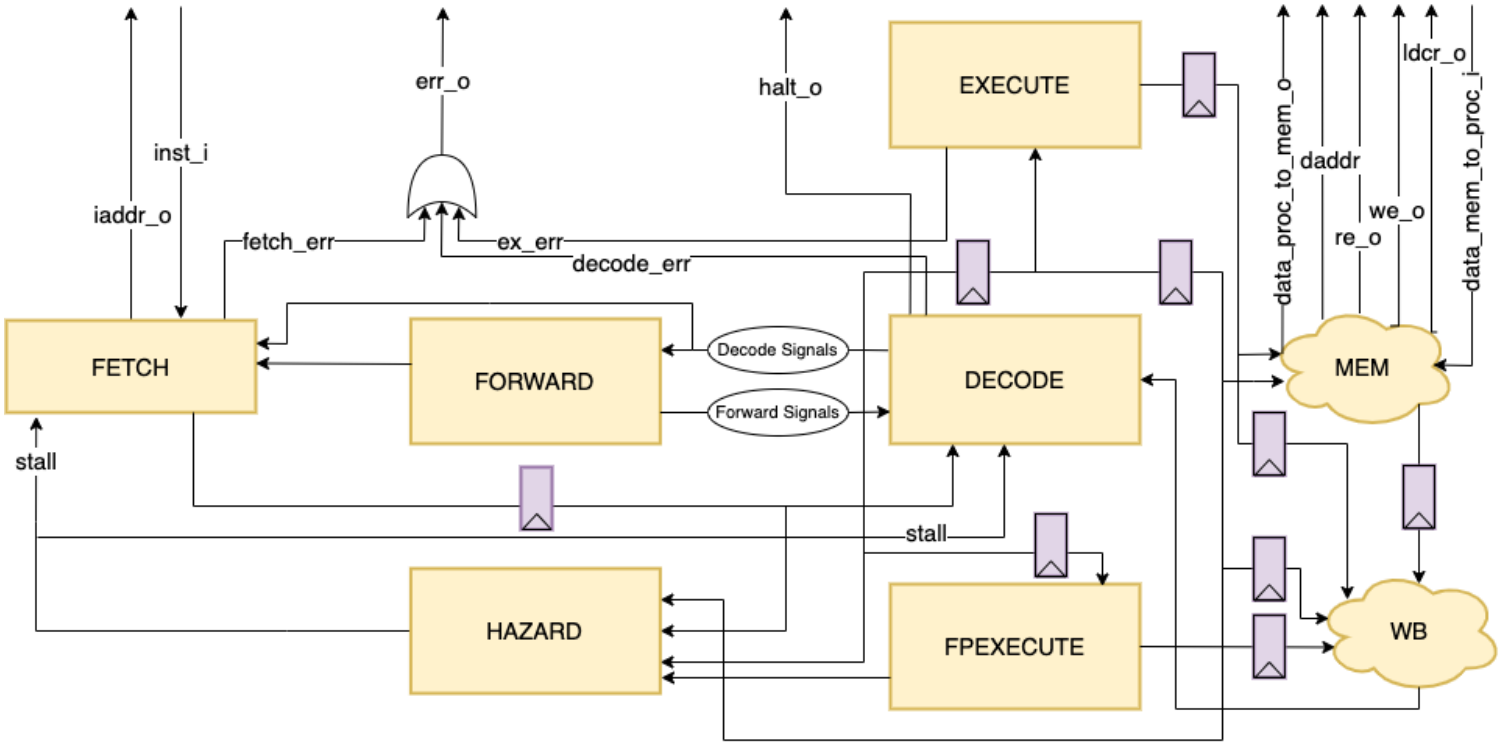
Signal	Input/Output	Width	Description
clk	Input	[0]	50MHz clock
RST_n	Input	[0]	FPGA Reset (Not Synchronized)
halt	Output	[0]	CPU encountered a Halt instruction. Used for debug.
RX	Input	[0]	UART RX line
TX	Input	[0]	UART TX line
VGA_CLK	input	[0]	Timing for the VGA unit.
VGA_BLANK_N	output	[0]	Asserted 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_RGB	output	[23:0]	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.

1.2. CPU

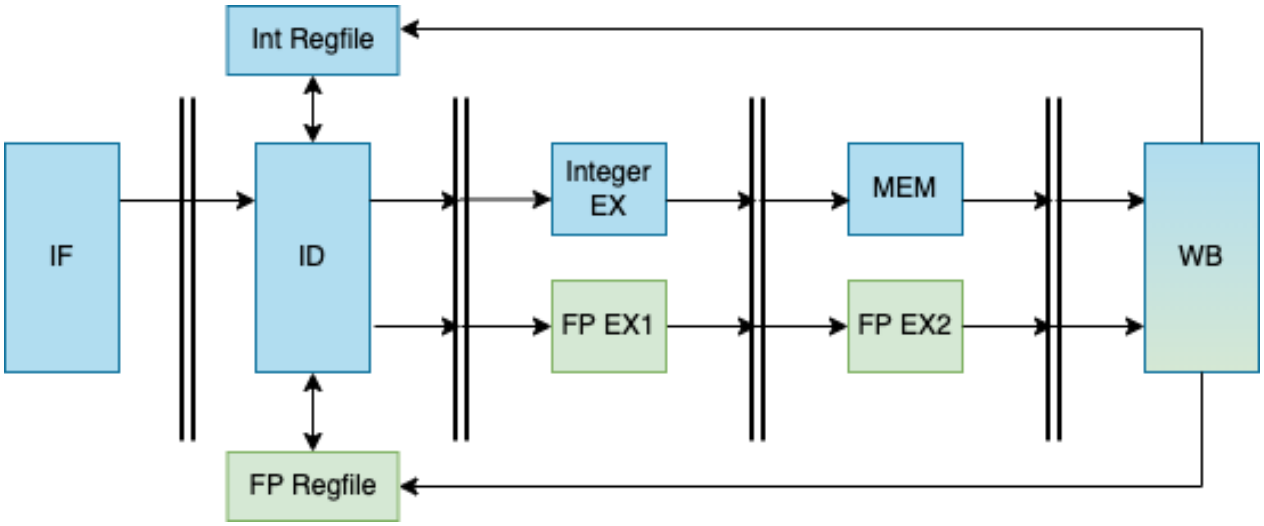
The CPU contains blocks and pipeline flops for the FETCH, DECODE, EXECUTE, FPEXECUTE, MEM and WB stages. It also has separate blocks to handle forwarding and hazard detection logic.

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 -



1.2.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	[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_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

1.2.1. Fetch

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

1.2.1.A Registers

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

1.2.1.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
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.

1.2.2. Decode

Decode block performs the following functions:

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

1.2.2.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.

1.2.2.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
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]	Register Write
MemWrite	Output	[0]	Memory Write
MemRead	Output	[0]	Memory Read
InstFmt	Output	[1:0]	00 - I-Format 2 01 - I-Format 1 10 - R-Format

			11 - J-Format
MemToReg	Output	[0]	Write to register from mem
AluSrc	Output	[0]	Immediate value is a source
AluOp	Output	[4:0]	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]	Zero extend
Exc	Output	[0]	IRQ seen
Halt	Output	[0]	HALT seen
FPInst	Output	[0]	FP Instruction seen
FPIntCvtReg	Output	[1:0]	FPIntCvtReg[0] - Write to Int Regfile. Source FP Reg. FPIntCvtReg[1] - Write to FP Regfile. Source Int Reg.
InstMemRead	Output	[0]	LDCR seen
ctrl_err	Output	[0]	Non supported instruction seen
decode_err	Output	[0]	Tied to 0

1.2.3. Execute

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

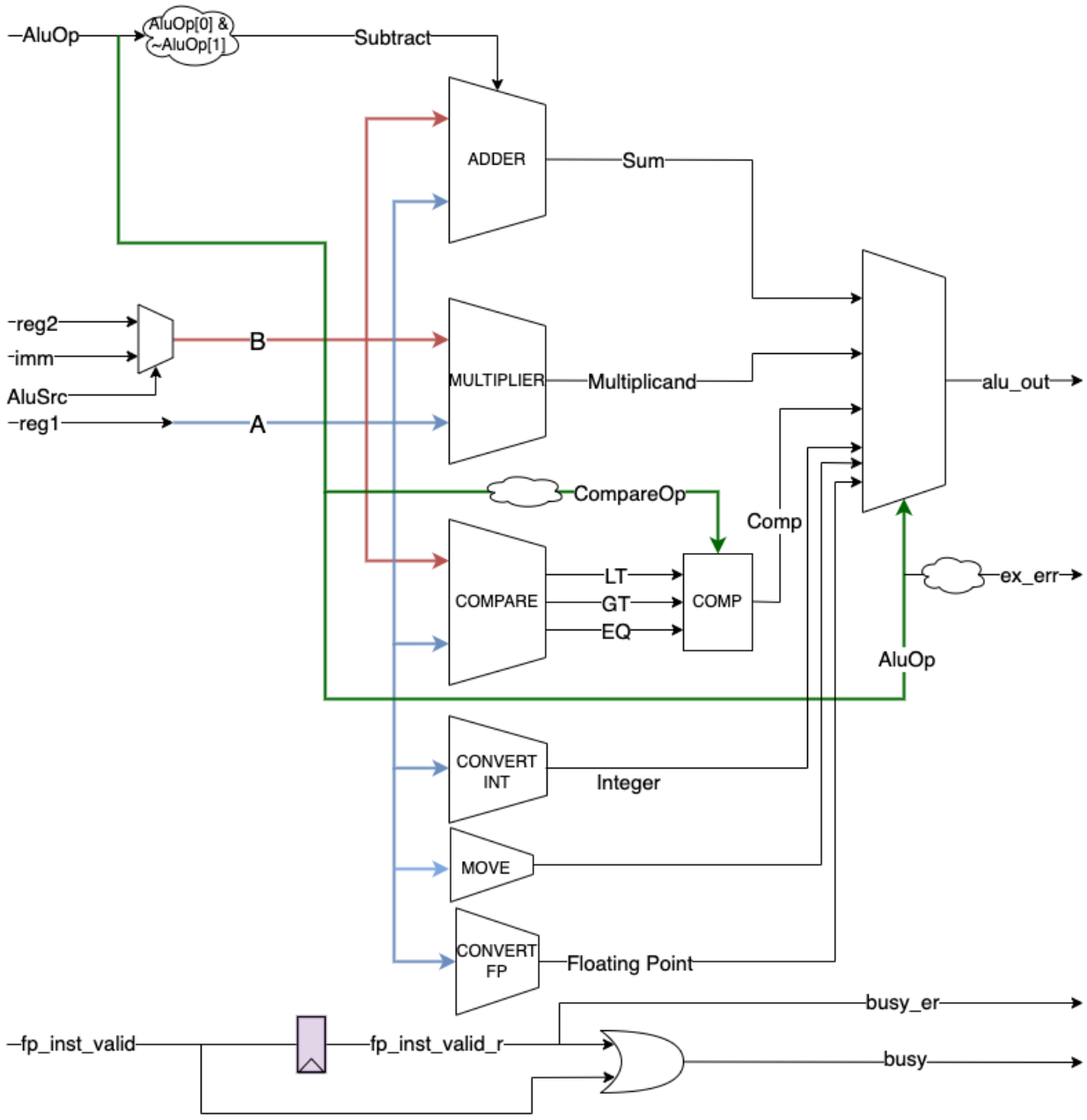
1.2.3.A Interface

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
ex_err	Output	[0]	Unsupported ALU operation

1.2.4. FP Execute

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



1.2.4.A Registers

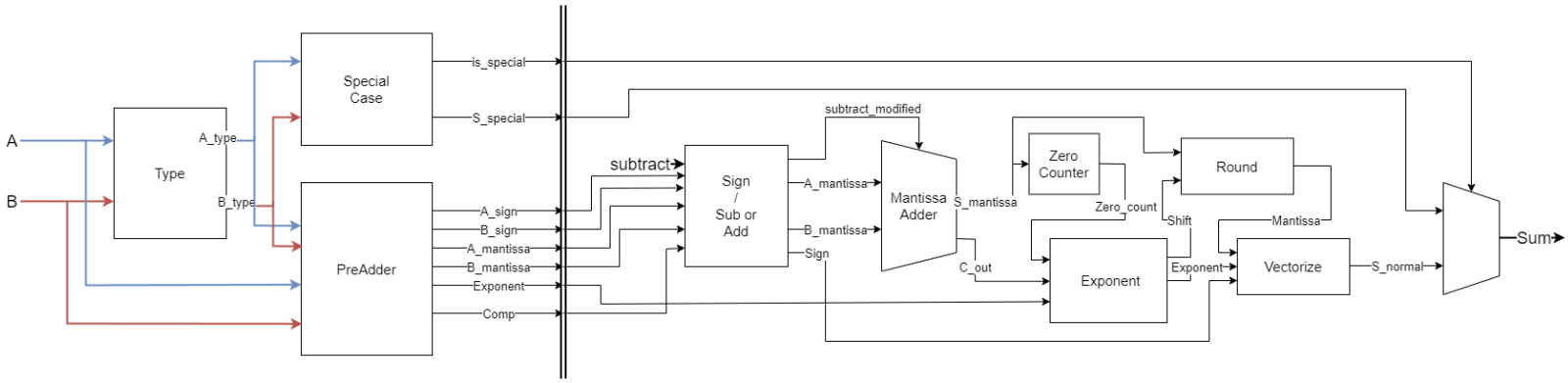
Register	Description
fp_inst_valid_r	Pipelined signal which determines if the FP EX is busy and early busy release.

1.2.4.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

1.2.4.1 FP Adder

Pipelined after the preadder.



1.2.4.1.A Registers

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

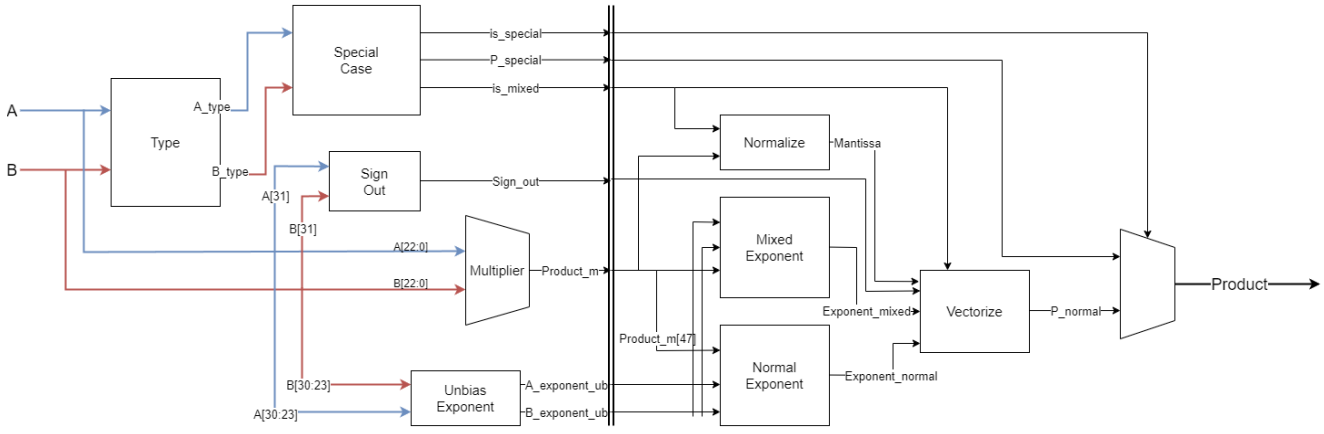
S_special	Pipelined signal which holds the sum when there is a special case.
A_sign	Pipelined signal which holds the sign of A.
B_sign	Pipelined signal which holds the sign of B.
A_mantissa	Pipelined signal which holds the mantissa of A
B_mantissa	Pipelined signal which holds the mantissa of B
Exponent	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

1.2.4.1.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

1.2.4.2 FP Multiplier

Pipelined after mantissa are multiplied.



1.2.4.2.A Registers

Register	Description
is_special	Pipelined signal which determines if the product is a special case.
P_special	Pipelined signal which holds the product when there is a special case.
is_mixed	Pipelined signal which determines if the product is a mixed case.
Sign_out	Pipelined signal which holds the sign of the product.
Product_m	Pipelined signal which holds the product of the mantissas (48 bits).
A_exponent_ub	Pipelined signal which holds the un biased exponent of A.
B_exponent_ub	Pipelined signal which holds the un biased exponent of B.

1.2.4.2.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

1.2.5. 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.

1.2.5.A. Interface

Signal	I/O	Width	Description

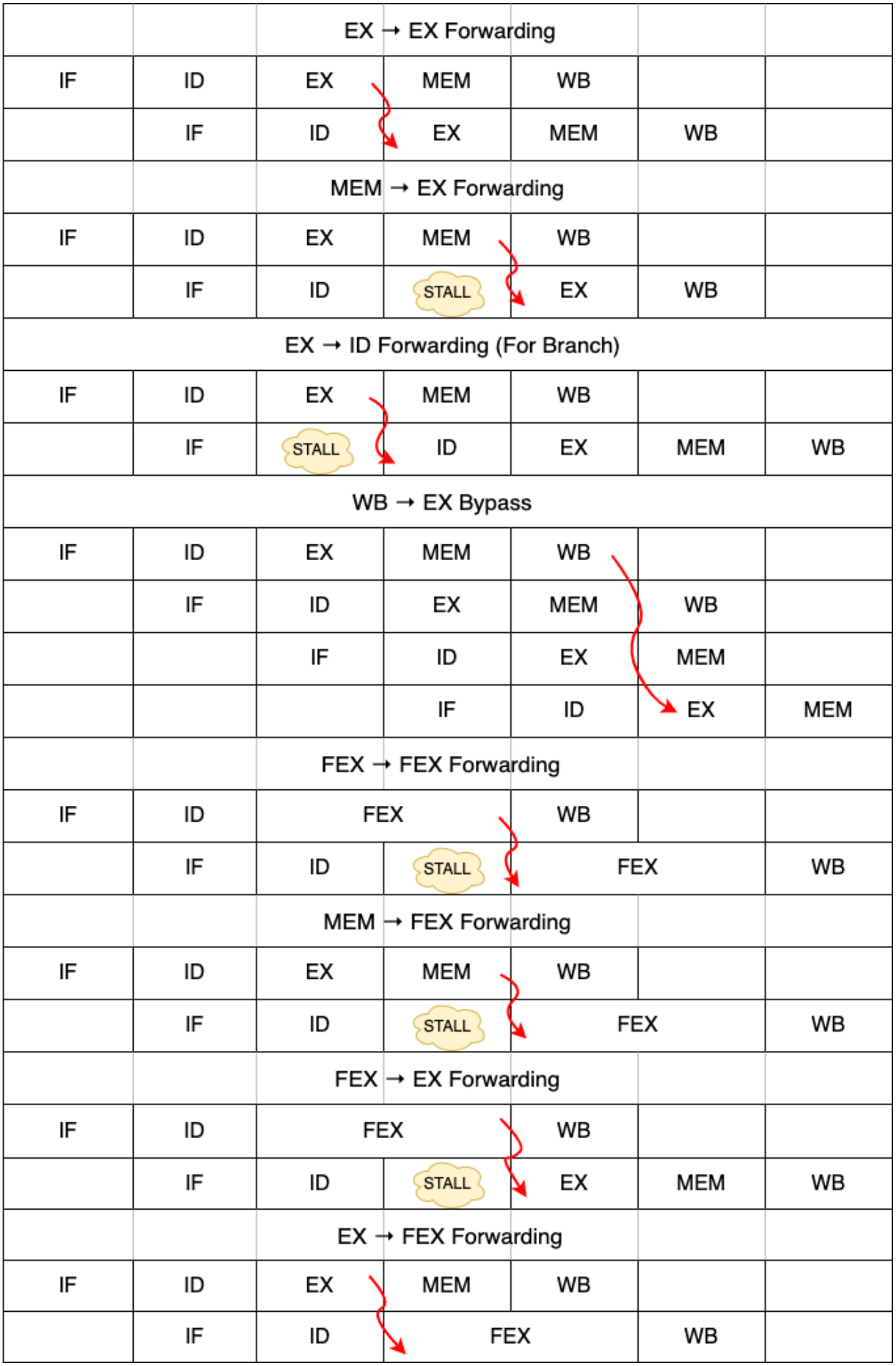
IF_ID_reg1	Input	[4:0]	Fetched instruction register A index
IF_ID_reg2	Input	[4:0]	Fetched instruction register B index
ID_EX_regw	Input	[4:0]	Decoded instruction register write index - out of decode stage
EX_MEM_regw	Input	[4:0]	Executed instruction register write index - out of execute stage
ID_EX_ctrl_regw	Input	[0]	Is decoded instruction writing a register - out of decode stage
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
ID_EX_is_load	Input	[0]	Load instruction decoded
FEX_busy	Input	[0]	FP Execute is busy
ID_FEX_is_fp_ex	Input	[0]	FP instruction decoded
stall	Output	[0]	Stall when hazard detected

1.2.6. Forward

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

1.2.6.A Pipeline

The following pipeline diagrams show all the 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
FEX → EX Forwarding
IF ID FEX WB
IF ID STALL EX MEM WB
EX → FEX Forwarding
IF ID EX MEM WB
IF ID FEX WB

1.2.6.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
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
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
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)
bypass_reg1	Output	[0]	Bypass source A from WB -> EX
bypass_reg2	Output	[0]	Bypass source B from WB -> EX
ID_FEX_fp_inst_valid	Input	[0]	Is the instruction decoded an FP instruction?

ID_FEX_reg1	Input	[4:0]	Decoded instruction FP register A index
ID_FEX_reg2	Input	[4:0]	Decoded instruction FP register B index
ID_FEX_ctrl_FPIntCvtReg	Input	[1:0]	Which register class is being written? Int for FP?
FEX_WB_ctrl_regw	Input	[0]	Is register being written - out of FEX2 stage
frwd_WB_FEX_opA	Output	[0]	Forward source A from FEX2 -> FEX1 stage
frwd_WB_FEX_opB	Output	[0]	Forward source B from FEX2 -> FEX1 stage
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

1.2.7. MEM

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

1.2.8. Write Back

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

1.3. RST

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

1.3.A. Registers

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

1.3.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.

1.4. IMEM

Dual-port 32kB Instruction memory.

1.4.A. Registers

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

1.4.B. Interface

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

1.5. DMEM

Single-port 16kB Data memory.

1.5.A. Registers

Register	Width	Description
mem_r [4096]	[31:0]	Instruction Memory
rdata_r	[31:0]	Data Read

1.5.B. Interface

Signal	Input/Output	Width	Description
clk	Input	[0]	
addr_i	Input	[31:0]	Address to read from data memory.

we_i	Input	[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.

1.6. SPART

Memory-mapped SPART peripheral. SPART interface will handle keyboard input and graphics output. This will be used as a fall back option if VGA doesn't work out.

1.6.A. Registers

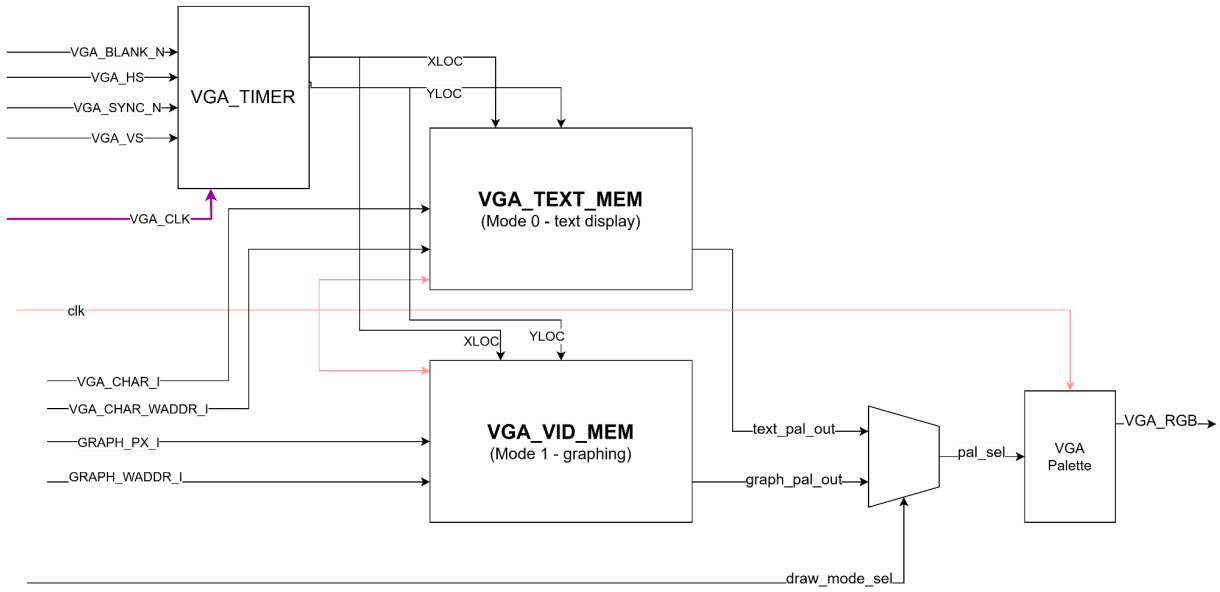
Register	**Address (range)**	**Description**
TX_BUF_RW_REG	0x0000C004	TX/RX buffer. Read is a read from RX circular queue, write is write to TX circular queue.
STATUS_REG	0x0000C005	Status Register (read only)
DB_LOW_REG	0x0000C006	DB (low) Baud rate division buffer low byte
DB_HIGH_REG	0x0000C007	DB (high) Baud rate division buffer high byte

1.6.B. Interface

Signal	**Input/Output**	**Width**	**Description**
databus	inout	[7:0]	An 8-bit, 3-state bidirectional bus used to transfer data and control information between the

			Processor and the SPART
ioaddr	input	[1:0]	A 2-bit address bus used to select the particular register that interacts with the databus during an I/O operation
iorw_n	input	[0]	Determines the direction of data transfer between the Processor and SPART. For a read (iorw_n=1), data is transferred from the SPART to the Processor and for a write(iorw_n=0), data is transferred from the processor to the SPART
iocs_n	input	[0]	Active low chip select. Writes or reads to registers have no effect unless active

1.7. VGA GPU



The VGA GPU / driver is designed to both offer practical drawing options for a graphing calculator as well as 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 [will likely go unused]).
- Quarter-size line-drawing framebuffer (320x240) for visualizing equations. Each pixel has 4 bits of palette info, resulting in a 38.4kb buffer.

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

1.7.A. Registers

Reserved address spaces for VGA_VMEM_TEXT and VGA_VMEM_GRAPH are tentative.

Register	Address (range)	Description
VGA_VMEM_TEXT	0x0000D000-0x00011E40	VGA Text Mode framebuffer
VGA_VMEM_GRAPH	0x00012000-0x00016B00	VGA Graph Mode framebuffer

1.7.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	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

1.8. PS/2 Keyboard

Interface to the PS/2 keyboard I/O. Keystrokes handled via interrupts to the hardware, which is implemented with an interrupt line to the interrupt controller.

1.8.A. Registers

Register	Address (range)	Description
PS2_CONFIG_REG	TBD	Configuring PS2.
PS2_KEY_PRESSED	TBD	Memory-mapped value that is read by the program to get the last key pressed.

1.8.B. Interface

Signal	Input/Output	Width	Description
MM_PS2_INPUT	input	[7:0]	Input from the control unit in case the PS/2 interface requires configuration.
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_KEY_PRESSED	output	[7:0]	Has the last key pressed.

1.9. 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.

1.10. PLL

1.10.A. Interface

Signal	Input/Output	Width	Description
refclk	Input	[0]	REF CLK
rst	Input	[0]	Reset
outclk_0	Output	[0]	Clk, 50 MHz
outclk_1	Output	[0]	VGA_CLK, 25MHz
locked	Output	[0]	PLL locked

2. Software

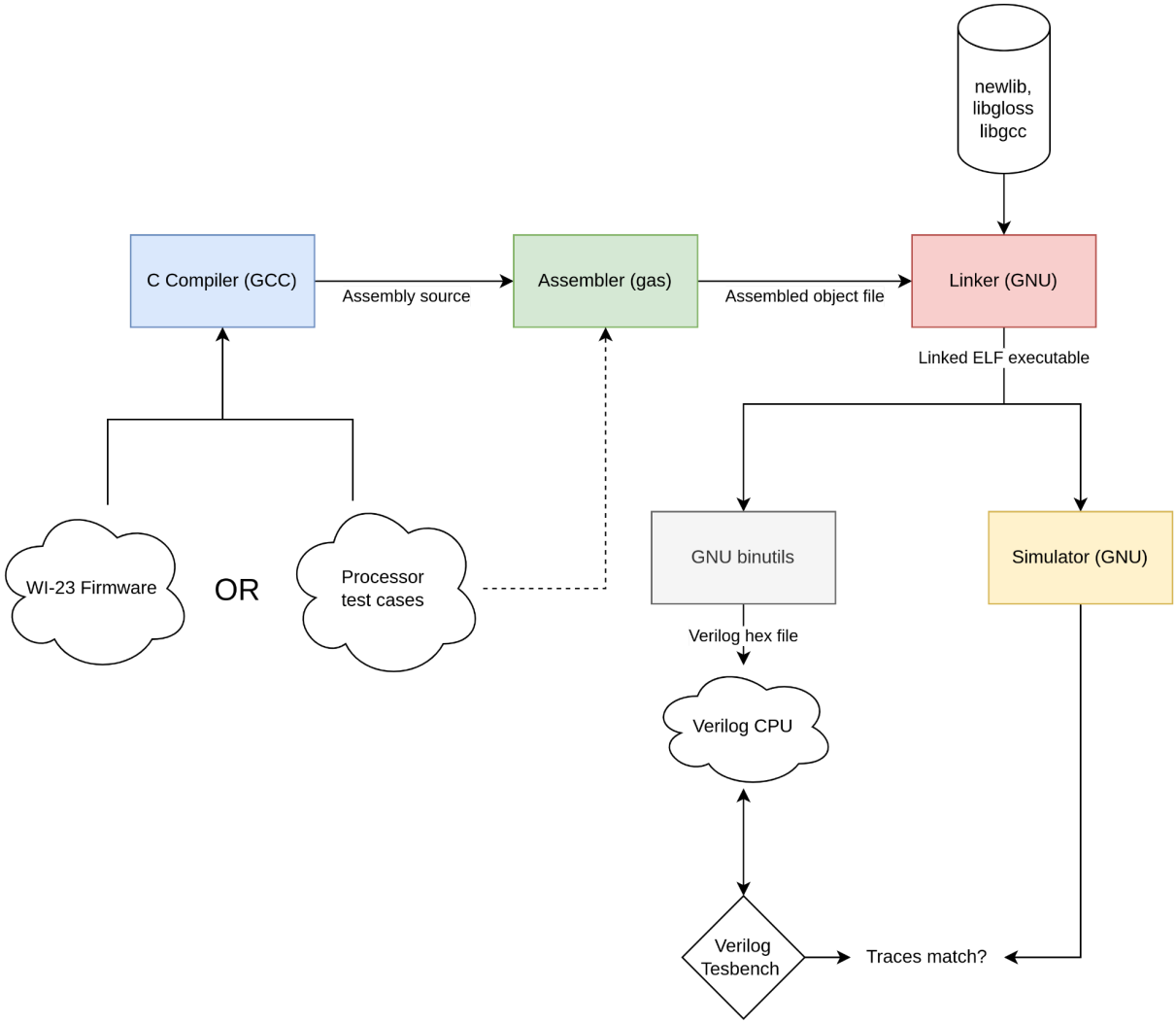
It is difficult for us to generalize what software your project entails. All projects involve the firmware running on the processor. This section should include a description of this software, and how it interfaces with the hardware elements.

Some projects also involve software external to the hardware system (learning software for ML, simulator for processor, compiler for processor, …). Please provide a brief description of this software and how it integrates or affects the entire system.

2.1. Toolchain

The diagram below outlines a comprehensive flow from C code to machine code, specifically from the context of verification. Creating a bitstream with the firmware built-in would work the same way only the Verilog would be synthesized instead of simulated.

Note that the simulator is not planned to support simulation models for the memory-mapped peripherals, and neither is the CPU's verilog testbench.



Below is a definition of the blocks in this diagram.

1. **WI-23 Firmware**: Firmware that will run on our processor on the DE1 board providing the calculator functionality
2. **Test cases**: Unit tests that can be run through the toolchain to facilitate verification. These can come in the form of C code or simpler assembly files, testing single instruction functionality at a time. Note: GCC already has a large suite of C test cases which we will take advantage of, but we will also write several assembly-level tests.
3. **Compiler**: C compiler using GCC.
4. **Assembler**: Assembler for WI-23 using the GNU as infrastructure.
5. **Linker**: GNU linker for WI-23.
6. **newlib/libgloss/libgcc**: Runtime libraries to be linked with compiled applications. newlib/libgloss comprise a libc for embedded systems, which is needed to run the testsuite (mainly due to the requirement of a C startup file crt0.S which goes in newlib.)

libgcc is a runtime library with support routines linked in by GCC when it tries to generate an instruction pattern that the target ISA doesn't have, for instance an integer multiply.

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 ran.
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.

2.2 Parser

The parser is the component of the component of the system that connects the PS/2 Keyboard with the computing software and the graphing software. The parser uses the memory mapped PS2_IRQ and PS2_DAT signals to read input from the user. Every time a new character is made available, it is added to the input string. The input is simultaneously echoed to the VGA monitor by writing the characters to the VGA_VMEM_TEXT address space. After the entire input is read, the string will be tokenized and passed to an implementation of the Shunting Yard Algorithm (https://en.wikipedia.org/wiki/Shunting_yard_algorithm) that converts the input to reverse-polish notation. This string of characters in reverse-polish notation is the final output of the parser and is passed to either the compute software or graphing software for processing.

2.3 Front-End Software

While the hardware is indeed an undertaking in of itself, the front-end is a non-trivial task and there will be significant effort to make the interface not only accessible, usable, and responsive, but even aesthetically pleasing.

In addition, the display routines for text and the menus (including wrapping) as well as the code for token parsing are both not trivial tasks either and will require significant development time.

2.3.1 The REPL Environment

The calculator has a REPL environment to do calculations, much like any other graphing calculator. This is the "main" mode of the graphing calculator, and is shown on boot. The user may press **F1** to navigate to this menu from other modes.

This REPL environment is shown in VGA text mode, with text input, output, and history displayed in a shell-like environment.

Upon boot, this message is shown:
"WI-23 GRAPHING CALCULATOR"

“>>”

With the “>>”, the user is prompted to enter in an equation. If the equation is too long for one line, it is autowrapped around the window edges.

Upon the ENTER key being pressed, the equation is evaluated, and the output will be shown on the next line, with another prompt being issued after.

The code that evaluates these equations is relatively simple. It is passed an array of tokens representing an equation in reverse-polish notation, and it iterates through the array, solving the equation element by element. If an error occurs, “error: <error msg>” will be printed on the output line.

2.3.2 The Equation Editor

Up to four lines (number arbitrary to the menu space, TBD) may be graphed at once. These use the same colors as the text menu, and thus there are 16 colors.

The equations are entered in a special window: **F2** may be pressed to navigate to this menu. While there should be an arbitrary limit to the token length of an equation, the plan is that if an equation is too long, the last n chars are shown in the equation line, where n is the char size of the line box display element.

In addition, settings for graph boundaries may be entered here, including:

- -x coordinate (leftmost x coordinate boundary for graphing)
- +x coordinate (rightmost x coordinate boundary)
- -y coordinate
- +y coordinate
- N-step

N-step is the parameter that determines the x-values to be evaluated by the equations entered in. This is the “granularity” of the graphing software, so to speak. There will be a max value to N, but this is TBD.

When the equations are graphed (see 2.3.3), the same code that is used to evaluate the equations in 2.3.1 is used to evaluate the equation at a set of various intervals (calculated by -x, +x, and N-step), generating graphing data.

2.3.3 Graphing Mode

Once equations are graphed, they will be graphed in this mode.

While a message to prompt the user to enter equations in the Equation Editor would be preferred, it would be non-trivial: either a custom bitmap image or a text mode overlay would

have to be used to draw the message on or over the text mode framebuffer, each which come with disadvantages.

It is TBD whether the graph will be a display element in another menu (specifically the equation editor: this is possible because it is 320x240, which is a fourth of the size of the VGA display resolution), or displayed at double-size full resolution in its own menu. If the latter is chosen, **F3** may be pressed to navigate to this menu.

The lines will be drawn in software entirely (even with hardware support, at the bare minimum an integer multiplier and divider would be needed for Bresenham's algorithm, which we don't want to include in the hardware). Since we have access to floating point calculations, we may use a more sophisticated line rasterization algorithm.

2.3.4 Settings

While there will be predefined constants for operations, it may be useful to have the opportunity for the end user to be able to configure these: therefore, a settings menu reachable by **F4** is planned.

Settings could potentially include:

- Maximum Evaluation Time (Since interrupts have been shelved, the only way to break out of a calculation that takes too long is recording how long it has taken and aborting if it reaches this threshold. This should be configurable.)
- Potentially all the settings in the Equation Editor
- REPL history buffer size / toggle history

2.4 VGA Menu Visualizer

Due to the lack of software to make colored VGA text menus, a custom-written VGA text mode visualizer has been developed to make creating these menus much easier than punching the values in a hex file.

Figure 2.5: A screenshot of the VGA menu visualizer (in development).