The WI-23 General Purpose Graphing Calculator

Wisconsin Instruments

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

Project Proposal Document

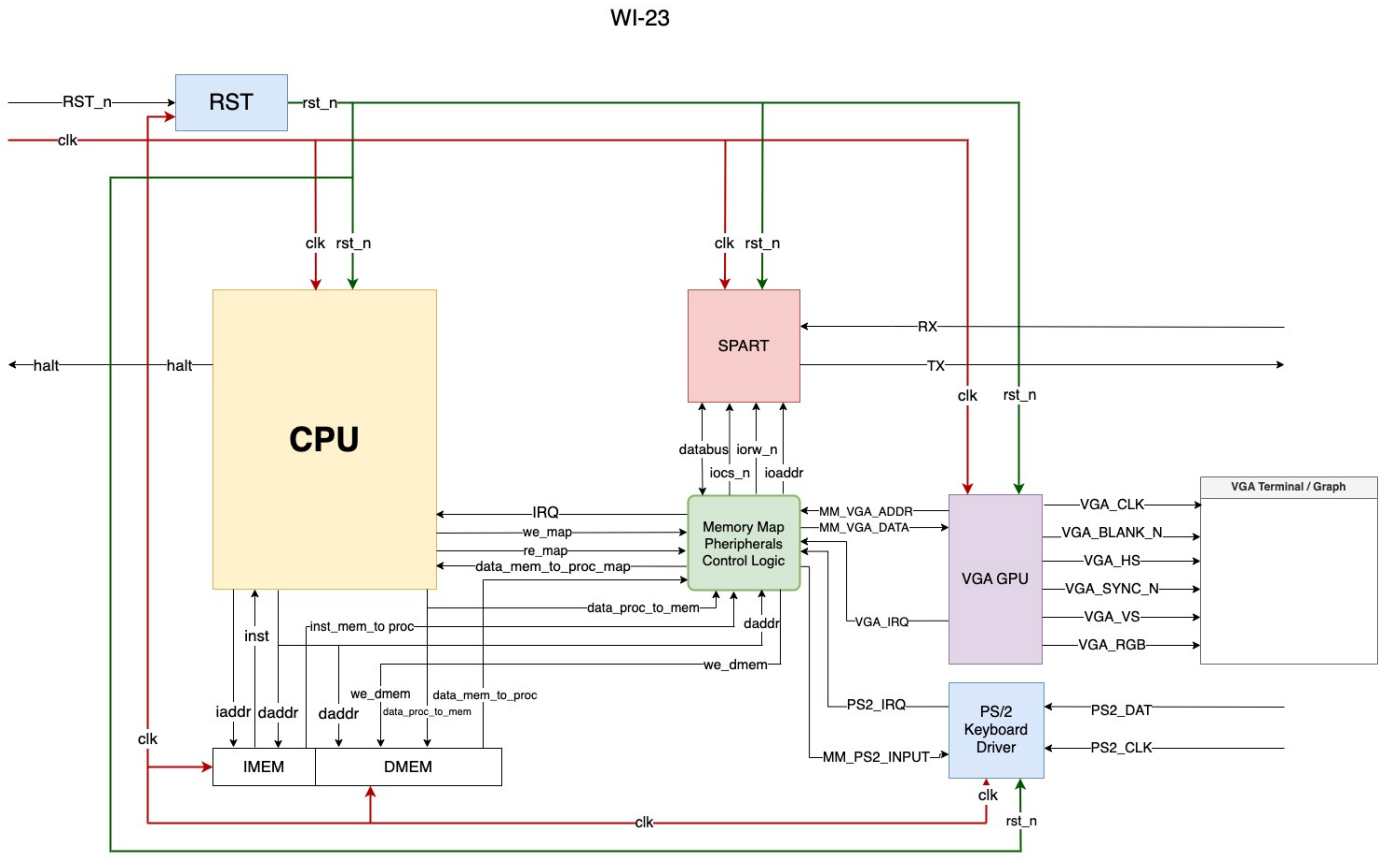
Table of Contents

1. Introduction

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 may also be graphed in an equation editor onto a VGA display.

2. Hardware Block Diagram



2.1 RST

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

Signal	Input/Output	Width	Description
clk	Input	[0]	Clock
RST_n	Input	[0]	Asynchronous reset from the FPGA.
rst_n	Output	[0]	Synchronized reset.

2.2 CPU

5 stage pipelined CPU.

2.2.A. Interface Signals

Signal	Input/Output	Width	Description
iaddr	Output	[31:0]	PC.
inst	Input	[31:0]	Instruction read from the instruction memory.
daddr	Output	[31:0]	Address to read/write to the data memory or the memory mapped peripheral. Address to read from instruction memory.
data_proc_to_mem	Output	[31:0]	Data write to the data memory.
data_mem_to_proc_map	Input	[31:0]	Data to the processor in response to the load instruction. This can be from the data memory or the memory mapped peripheral.
IRQ	Input	[1:0]	Interrupt the CPU for input from VGA and PS/2 Keyboard.
we_map	Output	[0]	Write enable to the data memory or the memory mapped peripheral.
re_map	Output	[0]	Read enable to the data memory or the memory mapped peripheral.

2.3 IMEM

Dual-port Instruction memory.

2.3.A. Interface Signals

Signal	Input/Output	Width	Description
iaddr	Input	[13:0]	PC.
inst	Output	[31:0]	Instruction read from the instruction memory.
daddr	Input	[13:0]	Address to read from IM for LDCR
inst_mem_to_proc	Output	[31:0]	Data read from IM for LDCR

2.4 DMEM

Single-port Data memory.

2.4.A. Interface Signals

Signal	Input/Output	Width	Description
daddr	Input	[12:0]	Address to read from data memory.
we_dmem	Input	[0]	Qualified write enable to data memory. Only asserted if not writing to a memory mapped peripheral.

data_proc_to_mem	Input	[31:0]	Write data to data memory.
data_mem_to_proc	Output	[31:0]	Read data from data memory.

2.5 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.

2.5.A. Memory Map

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

2.5.B. Interface Signals

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

2.6 VGA GPU

- Uses two draw modes (implying two separate framebuffers):
- 80x50 VGA text mode. The character set is stored in memory somewhere. Uses 16 bits per character, resulting in an 8kb buffer.
    - Calculator text is drawn to the screen via this mode (but also may be echoed to a debug terminal of the same width and height)
- Quarter-size line-drawing framebuffer (320x240) for visualizing equations. Each pixel has 2 bits of palette info, resulting in a ~19kb buffer (exactly 19200 bytes).
    - 00: Transparent
    - 01: Red
    - 10: Green
    - 11: Blue
- Some memory-mapped value can be written to or loaded from for changing VGA draw modes.
- Has a line to the interrupt controller (like any other peripheral)

2.6.A. Memory Map

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

2.6.B. Interface Signals

Signal	**Input/Output**	**Width**	**Description**
VGA_IRQ	output	[0]	Asserted when the VGA

			unit needs attention (potentially to signal to software that it is safe to draw to the framebuffer dring VBLANK).
VGA_CLK	output	[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.

2.7 PS/2 Keyboard

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

2.7.A. Memory Map

TBD.

Register	**Address (range)**	**Description**
PS2_CONFIG_REG	TBD	Configuring PS2.

2.6.B. Interface Signals

Signal	**Input/Output**	**Width**	**Description**
PS2_IRQ	output	[0]	Asserted when the PS/2 keyboard needs attention from the software (for

			registering key presses).
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.

2.6 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. Processor

The WI-23 processor is an enhanced version of the WISC-SP13 (although there are a few key differences, so the WI-23 instruction set is not a strict superset).

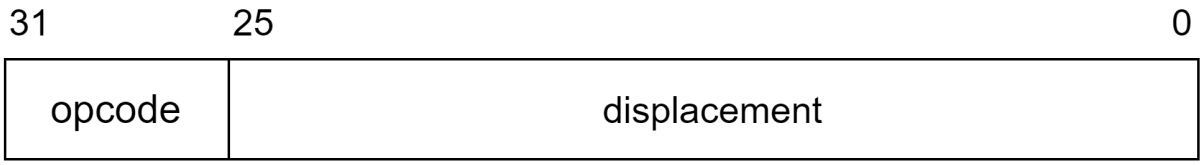
Design cues are taken from other successful recent RISC architectures (including RISC-V and MIPS prominently).

3.1. ISA Summary

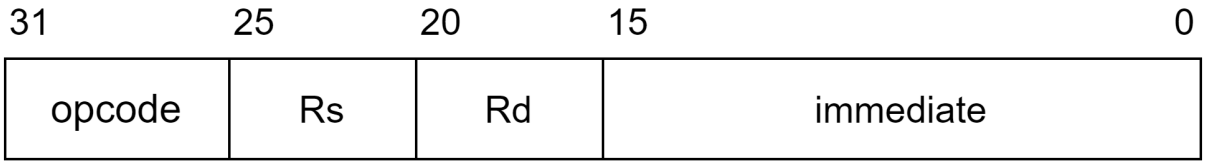
WI-23 is a full 32-bit architecture with a 32-bit datapath, 32-bit instructions and 32-bit addressing, with floating point support. All floating-point operations use IEEE-754 32-bit single precision binary representation. Only rounding mode is Round to Nearest, Ties to Even.

3.2. Instruction Formats

J-Format

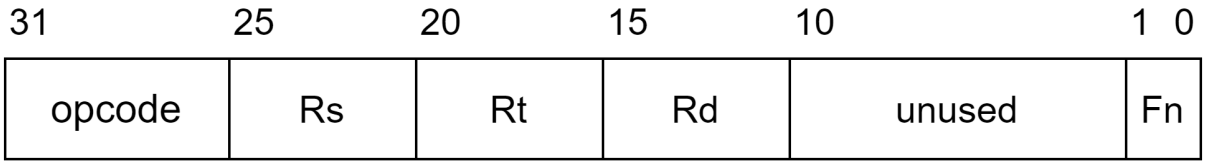


I-format



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.

3.3. Instructions

NOTE: All floating-point arithmetic operations are currently assumed to take 2 cycles. Floating-point rounding utilizes only one rounding mode: Round to Nearest, Ties to Even as defined by IEEE-754.

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	000010	XX	IRQ Imm		Produces a software interrupt. Hardware interrupts may be issued by the peripheral controller. Rd and Rs = don't-care.
I-format	000011	XX	LDCR Rd, Rs, imm	Rd <- IMem[Rs + I(sign ext.)]	Loads 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	
I-format	010001	XX	LD Rd, Rs, imm	Rd <- Mem[Rs + I(sign ext.)]	Load from data memory
I-format	010011	XX	STU Rd, Rs, imm	Mem[Rs + I(sign ext.)] <- Rd Rs <- Rs + I(sign ext.)	
R-format	011001	XX	BTR Rd, Rs	Rd[bit i] <- Rs[bit 31-i] for i=0..31	Rt field unused.
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	XX	SLT Rd, Rs, Rt	Rd <- (Rs < Rt) ? 1 : 0	
R-format	011110	XX	SLE Rd, Rs, Rt	Rd <- (Rs <= Rt) ? 1 : 0	
R-format	011111	XX	SCO Rd, Rs, Rt	Rd <- (carry out of Rs + Rt) ? 1 : 0	
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	111011	11	FDIV Fd, Fs, Ft	Fd <- Ft / Fs	Division by zero sets Fd to +INFINITY if Fs is +0, -INFINITY if Fs is -0.
R-format	111100	XX	FEQ Fd, Fs, Ft	Fd <- (Fs == Ft) ? 1 : 0	+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.)]	

R-format	100100	xx	FCLASS Rd, Fs	Rd <- FPStatus(Fs)	Very similar (if not identical) to RISC-V's floating point classification instruction. Takes a floating point register and dumps classification info to an integer register. 0: -inf 1: - 2: - (subnormal) 3: -0 4: +0 5: + (subnormal) 6: + 7+: NaN

Pseudo Ops / Assembler Directives

Syntax	**Equivalent To**	**Other Comments**
PUSH Rs	STU Rs, Sp, -4	32-bit words, stack grows downward.
POP Rd	LD Rd, Sp, 0 ADDI Sp, Sp, 4	
#include <filename.asm>		Copies the text of another .asm file and directly injects it at the location of the directive before assembling the source file.
#define LABEL <value>		Defines a variable in instruction memory with the label pointing to the variable itself.
#string LABEL <info>		Places a null-terminated string in instruction memory with the label pointing to the first character. Each character uses ASCII-standard 1-byte encoding. Use #vgastring for writing to the VGA output terminal instead.
#vgastring LABEL <info>		Places a null terminated string in instruction memory with the label pointing to the first character. Each VGA character contains 2 bytes of info.

		Attribute info may be set with a control code in hex. “\04hello \00world\n” defines a string with a red colored “hello” and a black colored “world”. See Understanding text mode - The intermezzOS Book for more details.

3.4. Register Sets

Register	ABI name	Description
Special Purpose		
r31	csr	Status register. Reserved for use with interrupts, overflow
f31	fcsr	Floating point status register. Reserved for overflow/divide by zero etc.
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
r27	gp	Global pointer
r0-r26	r0-r26	General-purpose registers.
General Purpose Floating Point		
f0-f30	f0-f30	General-purpose floating point registers.

ABI/calling convention details in general purpose registers might be subject to change during compiler development. As noted these do not affect hardware development.

3.5. Condition Codes

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

3.6. 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.

3.4. Immediate

Imm16 : 16-bit immediate.

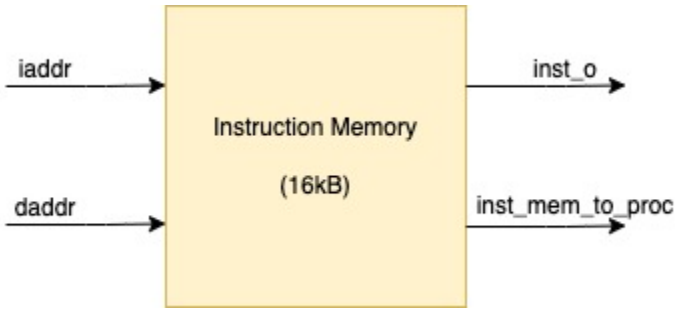
- Sign- or Zero- extended
- Used with all I-format instructions. Arithmetic, loads, stores, and branches.

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.

3.5. Harvard vs Von Neumann



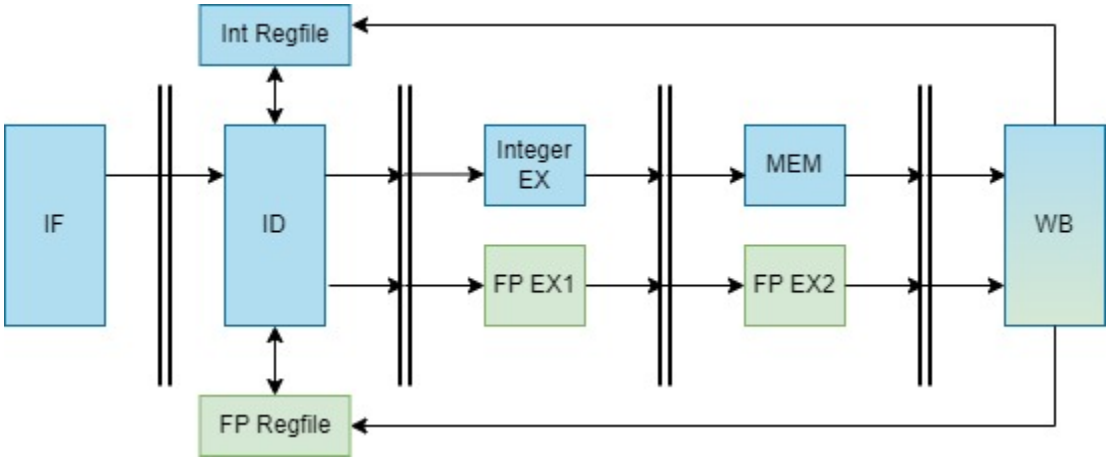
The WI-23 Graphing Calculator uses a Harvard architecture inherited from the WISC-SP13 architecture.

Instruction memory has 2-read ports. IM is read on the falling edge of the clock.

Port 0: Read 32-bit instruction

Port 1: Read 32-bit data (LDCR)

3.6. Special Features



Dual-issue pipeline implementation.

The following sections list the changes to the 552 five-stage pipeline processor.

3.6.A. Fetch

- Jump to an interrupt handler when an IRQ interrupt is received.
- Dual-port IM.

3.6.B. Decode

- Separate FP register file.
    - Register file is dual-ported (2-reads/1-write) similar to integer register file.
- Bypass logic for FP regfile.
- Stall FP instruction in decode if dependent on the result of a previous FP/Int instruction.
- A signal which says if the instruction being executed is FP or Int.

3.6.C. Execute

- New 2-cycle FP execute running in parallel with integer pipeline.
- A signal which says if the instruction being executed is FP or Int.
- FP execute unit is busy till after the completion (2-cycles). Stall instructions in decode till the FP EX is busy.

3.6.D. Execute

- A signal which says if the instruction being executed is FP or Int.
- FP loads/stores use the integer pipeline.

3.6.E. WB

- Write to the new FP register file if the instruction was FP.

3.7. Co-processors

None (the Floating Point unit is contained in the CPU itself)

3.8. Other architectural features

FP execute unit (described in 3.6).

4. Software Blocks

4.1. Assembler/Compiler

The assembler has macro support not found in the original WISC-SP13 assembler including string and constant defines, includes, and pseudo-ops such as PUSH and POP. We will extend this assembler codebase to support these features.

There are plans to create a compiler for our ISA following a two week feasibility study carried out by Julien. If it proves to be too much work, it will be scaled back.

4.2. Simulator

If a simulator is used in your project, explain the inputs and outputs for your simulator and how that will help in your project.

4.3. Application

The application layer of the project is reminiscent of other graphing calculators.

- REPL Mode (F1): An 80x50 text terminal that allows the user to input an equation on a line.
    - When ENTER is pressed, the equation is evaluated, either printing out a result or causing an error message to be displayed (divide by zero, etc).
- Equation Mode (F2): Up to three equations may be entered in this graphing window.
    - They can either be used in REPL mode by using Y1(), Y2(), or Y3(), or in Graph Mode.
- Graph Mode (F3): Equations defined in equation mode are drawn to the screen.
- Additionally, input/output may be configured to utilize the SPART.

5. Division of Labor

Provide a table indicating high level tasks that different team members will be responsible for. You will not be held to this initial division of labor, but we need to know you have at least give it some thought.

Tasks	**Who owns it**	**Comments**
Compiler	Julien	Design: ● Uses LLVM to make a compiler for C into our ISA. Validation: ● Use LLVM's unit test framework to write regression tests for the ISA backend.
Parser	Elan	Design: ● LR parser written in C (so comes after the Compiler task)
Fetch	Design: Madhav Validation: Madhav	Design: ● 32-bit instruction fetch. ● 32-bit data fetch for LDCR. ● IRQ support to jump to instruction handler. Validation: ● None.
Decode	Design: Madhav Validation: Madhav (Julien if compiler doesn't work out)	Design: ● Move to 32-bit instruction decode. ● LDCR and FP instruction decode. ● Control signals to drive a) FP execute, b) FP loads/stores in Integer pipeline, c) LDCR from IM ● Bypass logic for FP registers. ● Stall logic for FP. Validation: ● Validate that old instructions work after the decode change. ● Self check control signals for FP and LDCR. ● Validate stall of an FP instruction when a) FP execute dependant b) FP load dependant
Execute	Design: Ayaz Validation: Ayaz (Julien if compiler doesn't work out)	Design: ● 2-cycle FP execute unit ● Busy signal assertion. ● FP flags Validation: ● Self check every FP instruction output. ● Create a stall condition when FP EX is busy.
Mem	Design: Madhav Validation: Madhav	Design: ● 32-bit read/write from DM. ● MUX output from IM and DM (LDCR requirement)

		Validation: ● Check that MUX is correct.
Write Back	Design: Madhav Validation: Madhav	Design: ● FP registers write both from register and integer pipeline. Validation: ● Self check register value written.
Top-level logic	Design: Aidan Validation: Aidan	Design: ● Stitching logic for all external units into / from the processor. Validation: ● Project-level operation unit tests.
Assembler 2.0	Elan	Design: ● Increased support for macros etc to make life easier (until we ostensibly have a compiler) Validation: ● Test .asm out is as expected from an input file
VGA Graphics Processing Unit	Aidan	Design: ● Uses two draw modes (implying two separate framebuffers): ● 80x50 VGA text mode. The character set is stored in memory somewhere. Uses 16 bits per character, resulting in an 8kb buffer. ○ Calculator text is drawn to the screen via this mode (but also may be echoed to a debug terminal of the same width and height) ● Quarter-size line-drawing framebuffer (320x240) for visualizing equations. Each pixel has 2 bits of palette info, resulting in a ~19kb buffer (exactly 19200 bytes). ○ 00: Transparent ○ 01: Red ○ 10: Green ○ 11: Blue ● Some memory-mapped value can be written to or loaded from for changing VGA draw modes. ● Has a line to the interrupt controller (like any other peripheral) Validation: ● Test various equations out in software and make sure the hardware has the expected result. ● Ensure VGA text mode is correctly implemented via testing every character combo.
PS/2 Keyboard Input	Aidan	Design: ● Key strokes handled via interrupts to the hardware, which is implemented with an interrupt line to the interrupt controller.

		Validation: ● Test each key press in our calculator software / simulate presses in software with IRQ
Front-end	Aidan	Design: ● Has support for: ○ Standard calculator REPL ○ Input for functions to be graphed in a separate window ○ Graph viewer Validation ● Test responsiveness from peripherals, test parsing