8-bit Microprocessor

Anderson Hsieh

February 2024

Contents

1 Overview

1.1 What is WF8?

WF8 is an abbreviation for Waterfall 8 bits. It is a microprocessor project that consists of the ISA(instruction set architecture) along with hardware implementation using Verilog, targeting both FPGA and ASIC use cases.

1.2 CPU organization

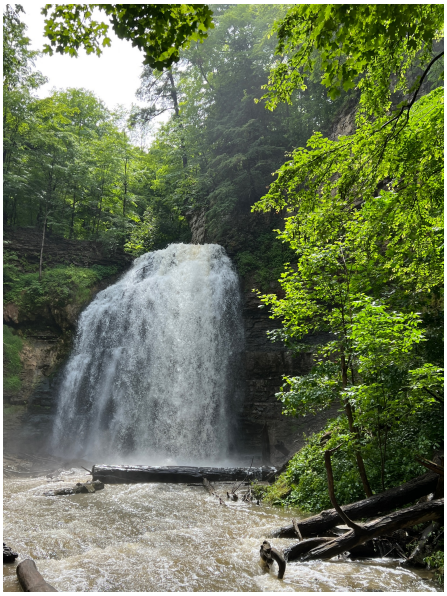
Generally, there are three types of CPU organizations

1. Stack-based
2. Accumulator-based
3. General registers

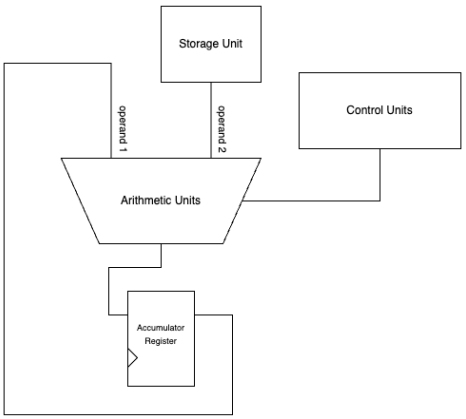
Most CPUs nowadays use general register-based organization, which is intuitive for assembly programmers. But to address as many instructions with the least amount of bits, while keeping the instruction length just 8 bits, I decided that accumulator-based organization would be the most suitable.

The accumulator-based organization uses the accumulator register as an operand to every instruction, which means in each instruction we only need to specify one operand register. For example, in RSIC-V ISA, the "ADD" instruction takes 3 operands registers: "add reg_a, reg_b, reg_c". This assembly instruction adds the values in reg_b and reg_c and stores the result in reg_a. In my design with the accumulator-based CPU organization, I can simply specify "add reg_a" as an instruction, and that implies adding value in reg_a to the accumulator register and storing the result in the accumulator register.

This is also why it's given the name "Waterfall". Like water flowing down a cliff and accumulating in the pond, the accumulator register holds the results of all calculations.



(a) Tiffany Falls in Ontario, Canada



(b) General data/control flow of WF8

Figure 1: Waterfalls

1.3 Inspiration and Credits

I learned a lot from Design a CPU by Ross McGowan on Udemy and Computer Architecture course at the University of Waterloo where I implemented a 5-stage pipelined RISC-V core. Those courses helped me visualize the possible implementation of certain instructions and inspired my design expansions. Shout out to Ross McGowan and my computer architecture professor, THANK YOU!

2 ISA

2.1 Overview

The WF8 ISA is designed to be minimal while being able to support most operations. Remember again that everything surrounds the accumulator register, in almost every instruction it is either one of the operand or the target register, or even both. Please see the figures below for a full list of instructions and their functionality.

instruction	op_code[3	op_code[2	op_code[1	op_code[0	utility bit	bit[2]	bit[1]	bit[0]	Name
add	0	0	0	0	X	reg_a[2]	reg_a[1]	reg_a[0]	register add
addi	0	0	0	1	imm[3]	imm[2]	imm[1]	imm[0]	add immidiate
sh	0	0	1	0	X	reg_a[2]	reg_a[1]	reg_a[0]	register shift
shi	0	0	1	1	imm[3]	imm[2]	imm[1]	imm[0]	shift immidiate
not	0	1	0	0	X	reg_a[2]	reg_a[1]	reg_a[0]	unary not
and	0	1	0	1	X	reg_a[2]	reg_a[1]	reg_a[0]	binary and
or	0	1	1	0	X	reg_a[2]	reg_a[1]	reg_a[0]	binary or
xor	0	1	1	1	X	reg_a[2]	reg_a[1]	reg_a[0]	binary xor
cpy	1	0	0	0	0	reg_a[2]	reg_a[1]	reg_a[0]	copy accumulator register
cpypc	1	0	0	0	1	reg_a[2]	reg_a[1]	reg_a[0]	cpy program counter
lb	1	0	0	1	X	reg_a[2]	reg_a[1]	reg_a[0]	load byte
sb	1	0	1	0	X	reg_a[2]	reg_a[1]	reg_a[0]	store byte
jmpadr	1	0	1	1	X	reg_a[2]	reg_a[1]	reg_a[0]	jump address
jmpi	1	1	0	0	imm[3]	imm[2]	imm[1]	imm[0]	jump immidiate
blt	1	1	0	1	X	reg_a[2]	reg_a[1]	reg_a[0]	branch less than
bgt	1	1	1	0	X	reg_a[2]	reg_a[1]	reg_a[0]	branch greater than
beq	1	1	1	1	0	reg_a[2]	reg_a[1]	reg_a[0]	branch equal to
bneq	1	1	1	1	1	reg_a[2]	reg_a[1]	reg_a[0]	branch not equal to

Figure 2: ISA encoding

instruction	In presudo verilog	Description
add	reg_acc = reg_acc + reg_b	add value in reg_b to value in reg_acc, then store in reg_acc.
addi	reg_acc = reg_acc + imm[3:0]	add value in reg_acc with value in reg_b, then store in reg_acc. Immidiate value = -8 ~ 7
sh	reg_acc = reg_b > 0 ? reg_acc << reg_b : reg_acc >> (~reg_b + 1)	shift value in reg_acc by value in reg_b, then store in reg_acc.
shi	reg_acc = imm[3:0] > 0 ? reg_acc << imm[3:0] : reg_acc >> (~imm[3:0] + 1)	shift by an immidiate value, immidiate value is in 2s complement. Immidiate value = -8 ~ 7
not	reg_acc = ~reg_acc	unary NOT operation on reg_acc, then store in reg_acc
and	reg_acc = reg_acc & reg_b	bitwise AND operation between reg_b and reg_acc, then store in reg_acc
or	reg_acc = reg_acc | reg_b	bitwise OR operation between reg_b and reg_acc, then store in reg_acc
xor	reg_acc = reg_acc ^ reg_b	bitwise XOR operation between reg_b and reg_acc, then store in reg_acc
cpy	reg_b = reg_acc	copy value in reg_acc into reg_b
cpypc	reg_b = PC	copy current instruction address in PC into reg_b
lb	reg_b = M[reg_acc]	load 1 byte from address B in memory, then store the value in reg_b. B is the value in reg_acc
sb	M[reg_acc] = reg_b	store value in reg_b to address B in memory. B is the value in reg_acc
jmpadr	PC = reg_b	unconditionally jump to a specific address in memory
jmpi	PC = PC + imm[3:0]	unconditionally jump to a specific offset relative to program counter in memory
blt	PC = reg_acc < reg_b ? PC + 2 : PC + 1	jump to PC + 2 if value stored in reg_acc is less than value in reg_b, else go to PC + 1 which is the default next instruction. Note imm = 2
bge	PC = reg_acc > = reg_b ? PC + 2 : PC + 1	jump to PC + 2 if value stored in reg_acc is greater or equal to the value in reg_b, else go to PC + 1 which is the default next instruction. Note imm = 2
beq	PC = reg_acc == reg_b ? PC + 2 : PC + 1	jump to PC + 2 if value stored in reg_acc is equal to value in reg_b, else go to PC + 1 which is the default next instruction. Note imm = 2
bneq	PC = reg_acc != reg_b ? PC + 2 : PC + 1	jump to PC + 2 if value stored in reg_acc is equal to value in reg_b, else go to PC + 1 which is the default next instruction. Note imm = 2

Figure 3: Functionality of each instruction

2.2 Terminology

Some ISA implementation-specific abbreviations and terminology are listed below

- pc = Program counter, an 8-bit register.
- imm = immediate value, specified by the executing instruction
- reg_acc = the accumulator register
- reg_b = One of the 8 available registers specified to be an operand of an instruction. Note that reg_acc can be a reg_b as well

2.3 Registers

There are 8 registers in total, including one accumulator register named reg_acc. The rest of the registers can be referenced by a 3-bit address from 3'b000 to 3'b111. Note that reg_acc has the address 3'b111.

2.4 Number Format

All numbers specified by the programmer in the instruction and any number stored in any registers are all interrupted as 2s complement, which means 8-bit registers allow us to cover [-128, 127].

2.5 Memory

Memory is byte-addressable, which means an 8-bit address allows us to address $2^8 = 256$ locations. Memory size is 256 bytes.

2.6 Instructions Overview

All the supported instructions can be categorized into the following sections.

2.6.1 ALU Instructions

Fundamental mathematics operations, including *add*, *shift*, *inversion*, *and*, *or*, and *xor*. The execution of these instructions all go through the ALU. Note that subtraction is done by adding a negative number (2s complement), and shift right is done by shifting by a negative amount, default shift is left.

2.6.2 Copy Instructions

Includes *cpy* and *cpypc*, these instructions helps us read data out of the accumulator register and program counter.

2.6.3 Memory instructions

Includes *sb* and *lb*, which stands for store byte and load byte.

2.6.4 Jump Instructions

Includes *jmpadr* and *jmpi*. These are unconditional jumps. *jmpadr* copies a reg_b to program counter, *jumpi* adds an immediate value to current program counter.

2.6.5 Branch Instructions

Includes *blt*, *bge*, *beq*, *bneq*. These are conditional jumps that only move the program counter if the condition is met.

3 FPGA Implementation

3.1 Overview

In order to use the on-chip block ram of FPGA in most commercial FPGAs, which has one clock cycle delay on synchronous read operations, it makes more sense to implement a pipelined architecture with extra control logic to eliminate data hazards. Figure 4 shows the main blocks in the pipelined datapath and which pipeline stage each block exists in. Note that a lot of the control signals are ignored, this picture is only an aid to understanding the Verilog implementation easier.

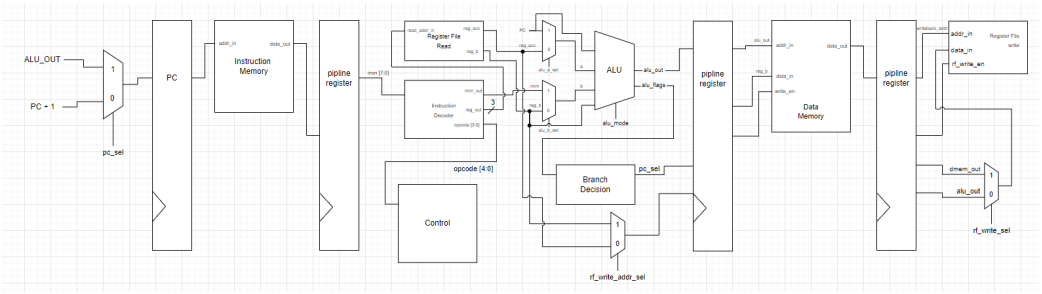


Figure 4: FPGA Pipelined Datapath Schematic

3.2 The Pipe-lined Datapath

My implementation is relatively similar to a typical RISC-V pipeline of **fetch - decode - execute - memory - write back** but because the WF8 ISA has such simple encoding of the instructions themselves, decoding logic should be relatively simple and small. A small decoding unit means shorter delays, so I combined it with the execute stage. This also makes eliminating data hazards easier, see section 3.5 for more detail. The following sections cover general functionality in each stage.

3.2.1 Fetch Stage : F

Use PC to read the current instructions. Note that synchronous read to the memory has one clock cycle delay, thus the output instruction is implemented through the pipelined registers as a bypass with no delay.

3.2.2 Decode + Execute Stage : DX

The instruction decoder generates the appropriate immediate value and register address to read from the register file. Note that reading to register file is an asynchronous read, value should be valid after a certain amount of delay. The ALU computes all arithmetic operations, note that during the execution branch instructions, the **XOR** unit in the ALU is reused for comparing the operands. see 3.5 for more detail.

3.2.3 Memory Stage : M

Write back to the data memory. The data memory block in the same address space as the instruction memory, aka size of data memory + size of instruction memory = 256 bytes. Data memory output does not go through the pipelined register when entering the next stage (WB) because this doesn't have

3.2.4 Write Back Stage : WB

Write the result of the instruction back to the register file. Most of the time the target register is the accumulator register. Synchronous write with 1 cycle delay, see Data Hazard and Register Dependency section 3.5 for more detail.

3.3 Memory & Register File

The instruction memory and data memory are implemented in separate modules in Verilog, but they are really just 2 Block RAM of size 256 bytes each since each of them is 8 bits ($2 * 8 = 256$).

3.4 Hardware Resource Reuse Optimization

There are 2 major optimizations in the current FPGA version of the RTL model.

First is the reuse of the XOR comparator. The branch instructions skip the next instruction if conditions are met. The XOR element in the ALU is implemented so that it can also be used to recognize greater than and equal to with bitwise manipulations. That means we can reuse it for checking the condition for branch instructions without instantiating a separate branch

comparator block.

The second optimization is also around the branch instructions. While the XOR comparator in the ALU is being used for checking the condition, the adder can be used for adding a "2" to the current program counter, this allows us to not have to instantiate an extra adder for PC just for the branch instructions. To implement this, the instruction decoder for immediate value outputs a "2" to the ALU which is then passed to the adder.

3.5 Data Hazard and Register Depedency

There are 4 types of data hazards:

1. Read after Write (RAW)
2. Write after Read (WAR)
3. Write after Write (WAW)
4. Read after Read (RAR)

The accumulator register is being written to and read from very often. The solution to this is either implement bypassing between stages and inserting NOPs (no operation, addi 0). This is handled by the hazard detect module which sits in the Fetch stage and NOPs are inserted at the pipeline register between the Fetch and DX stage.

For example, if we try to solve the problem with only NOP insertions and no bypassing, under the following assembly code, there will have to be 2 NOPs inserted between ADDI and SHI. Because ADDI needs to finish and write to the accumulator register before SHI can read from it. In other words, DX stage of SHI must come after ADDI's WB stage.

```
addi 1  -  F  DX  M  WB
NOP     -     F   DX M  WB
NOP     -         F  DX M  WB
shi 1   -            F  DX M  WB
```

3.6 I/O

Added an extra read port to the data memory module for synchronous memory mapped I/O read.

4 ASIC Implementation

4.1 Overview

This design is inspired by the single CPU bus design in the Design a CPU by Ross McGowan on Udemy. The idea is that most of the instructions have 2 operands. One is always the accumulator register, and the other one is user-specified. All the data traveling between storage units flows on the CPU bus, and the output of computational units is written directly to the accumulator register. Figure 5 With this design in mind, all the instructions

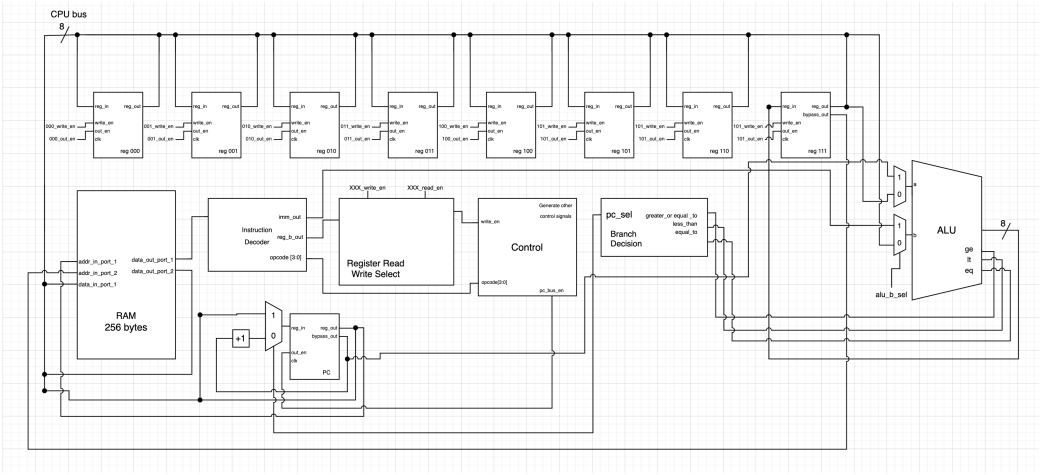


Figure 5: ASIC Datapath Schematic

will be executed in a single cycle. This forces us to slow down the clock frequency, which is a necessary trade-off for minimal resources synthesized.

4.2 Skywater130 and Tiny Tapeout

The goal of this ASIC implementation is to run the design through OpenLane flow and tape out through on TinyTapeout Multi Project Wafer. The Verilog code presented will be synthesized into standard cells in Skywater130 PDK and OpenRoad will do place and route of us.

4.3 CPU bus

The CPU bus is the core idea of this architecture. It is 8 bits and occupied by every storage unit in the datapath. All the storage units have an output enable signal that is asserted by the control unit. There will always be only one storage unit writing to the CPU bus at all times.

This single-bus architecture fits well with our accumulator-based CPU organization because the CPU bus can be hardwired to one of the inputs of the ALU, while the other one is hardwired to the accumulator register. While this is not the case due to some hardware optimizations, the idea remains that most of the time ALU input will be CPU bus (reg_b) and reg_acc.

4.4 Memory & Register File

Just like the FPGA implementation, the memory is 256 bytes and byte-addressable. The instruction and data memory shares the same physical memory. However, in the Verilog module the memory is described as one single block, without separating instruction/data memory. And the memory block has asynchronous reads, meaning that we can access the data in the same clock cycle. Ideally, this is synthesized into asynchronous SRAM cells in an ASIC.

The register file module includes all the registers and exposes separate 8-bit output for the accumulator register since it is needed as input to ALU for most of the instructions.

4.5 Hardware Optimizations

Similar to the FPGA implementation, the 2 major hardware resources reuse optimization applys here too. During branch instructions, the XOR unit for the "xor" instruction is being reused for branch comparisons, while the adder is being used to add a decimal 2 to the program counter. This is why there are still multiplexers for the ALU inputs. Otherwise inputs could've been hardwired to reg_acc and reg_b (the CPU bus).

4.6 I/O

Added an extra read port to the 256 byte memory for asynchronous memory mapped I/O read.

5 Simulations and Testing

5.1 Verilator and Cocotb

Currently setting up Verilator and Cocotb for simulation and testing. Work in progress.

5.2 Single Instructions

There will be short test cases to verify that each instruction works using the memory mapped I/O. Work in progress.

5.3 Benchmarks

Potentially small programs like bubble sort, Great Common Divisor, Tree Traversal, etc. Work in progress.