Designing a CPU

In this document, you will find my attempt at designing a simple 16-bit non-pipelined CPU in VHDL, using Xilinx Vivado.

In a non-pipelined CPU with components like the register file, control unit, decoder, PC unit, ALU, RAM, and CPU, each instruction is executed sequentially, going through the entire instruction cycle before the next one begins. Here's a hierarchical summary of the CPU:

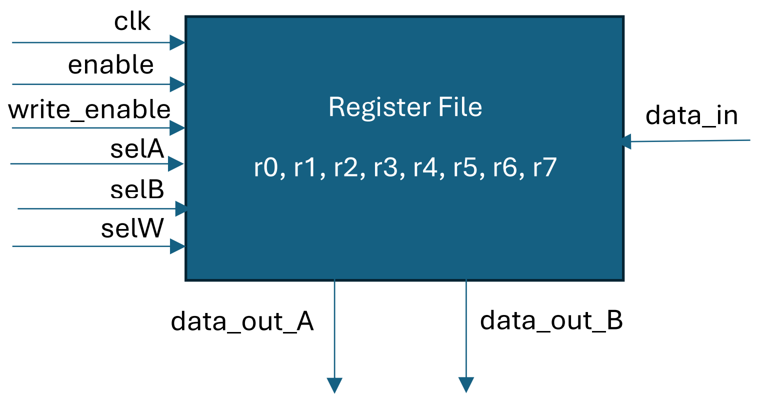
Top-Level Module: CPU

- **Submodules**:
  - **PC Unit**: Provides instruction addresses and handles branching.
  - **RAM**: Stores instructions and data.
  - **Decoder**: Interprets instructions and generates control signals.
  - **Control Unit**: Orchestrates each stage's execution in sequence.
  - **Register File**: Holds operands and stores results.
  - **ALU**: Executes arithmetic and logic operations.

Part 1

I will create a register file which will come in the form of 8 registers of 16-bit values.

The register file stores small amounts of data that the CPU needs for fast access. It holds operands for instructions and stores the results of computations.



The clk and enable bit are inputs that control the basic operation of the register file.

We have 3 inputs that indicate what registers we want for the destination and two sources; selA and selB are the sources and the destination is selW. We also have write_enable that controls the writing functionality of the register file. data_in is of course the data that enters the register file and will be handled accordingly to the sel inputs.

Part 2

Now we need to code the ISA (Instruction Set Architecture). The ISA of a CPU defines the set of operations that can be performed, and what data types the CPU can handle. It explains timing, restrictions and sometimes hardware bugs that may occur during normal operation. The operations are defined along with machine language, opcodes, instruction forms, register set and execution latencies. Simply put, it's the 'bare metal' interface to the operation of the processor.

Why do we need to define an ISA? Because it has direct consequences for the instruction decoder, the ALU and the register file.

To keep things simple, I will use a fixed instruction length, meaning 16 bits. Each instruction will always be 16 bits long, regardless of what it does. I have decided 16 bits because our machine is a 16-bit machine.

You may ask yourself, how can we fit all the information we need and the mathematical operation itself into those 16 bits? To answer that, let's look at our CPU and the operations which we want it to do. Keep in mind that this is a basic CPU which will handle basic integer operations and other useful functions.

Operation	Function
Add	D = A + B
Subtract	D = A – B
Bitwise Or	D = A or B
Bitwise Xor	D = A xor B
Bitwise And	D = A and B
Bitwise Not	D = not A
Read	D = Memory[A]
Set	Memory[A] = D
Load	D = 8-bit Immediate Value
Compare	D = cmp(A,B)
Logical Shift Left	D = A << B
Logical Shift Right	C = B >> A
Branch/Jump	PC = A Register or Immediate Value
Branch/Jump Conditionally	PC = Register if (condition) else nop

Above you can see the list of 14 basic operations which currently suit our needs. I will not go over them as I feel they are all easily understandable, except for the Load operation. You may ask yourself, why is C assigned an 8-bit immediate value instead of, let's say, 4 bits? Or even, 16 bits?

The answer is due to limited space in our ISA. Using 8-bit immediates in 16-bit architecture is commonly used due to optimization and space saving.

So, back to our operations. Due to having 14 operations we need 4 bits for an opcode.

Before continuing and assigning an opcode for each operation, let's take a closer look at the forms these operations take, and the inputs they require, to see what we can fit into any instruction in terms of additional information.

Instruction forms are the patterns instructions take. For example, the Add operation requires 3 registers to be supplied – 1 destination and 2 sources. However, Bitwise Not only requires 2 registers.

Now, let's get into the specifics.

Terms of the instructions:

R = register

U = unused

rD = destination register

rA= first operand register

rB = second operand register

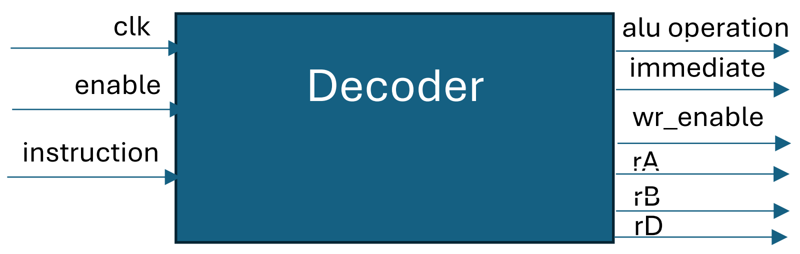
c = condition bit

I(8) = immediate data bits

Instruction	Form	Implementation	Writes Register	OPCODE
ADD	RRR	rD = rA + rB	c: 1/0 = signed/unsigned	0000
SUB	RRR	rD = rA – rB	c: 1/0 = signed/unsigned	0001
NOT	RRU	rD = not rA	c: N/A	0010
AND	RRR	rD = rA and rB	c: N/A	0011
OR	RRR	rD = rA or rB	c: N/A	0100
XOR	RRR	rD = rA xor rB	c: N/A	0101
LSL	RRR	rD = rA << rB	c: N/A	0110
LSR	RRR	rD = rA >> rB	c: N/A	0111
CMP	RRR	rD = cmp(rA, rB)	c: 1/0 = signed/unsigned	1000
B	UI(8)	PC = rA or 8-bit immediate	c: 1/0 = rA/8-bit immediate	1001
BEQ	URR	PC = rA conditional rB	c: N/A	1010
IMMEDIATE	RI(8)	rD = 8-bit immediate	c: 1/0 = upper/lower 8-bits	1011
LOAD	RRU	rD = memory(rA)	c: N/A	1100
STORE	URR	Memory(rA) = rB	c: N/A	1101

Below is the instruction layout and the block of the decoder:

Form	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RRR	C	opcode				rD			rA			rB			U	
RRU	C	opcode				rD			rA			U				
URR	C	opcode				U			rA			rB			U	
RI(8)	C	opcode				rD			8-bit immediate							
UI(8)	C	opcode				U			8-bit immediate							



As with any CPU, we need to define RAM (Random Access Memory).

This RAM will hold 256 addresses of 16 bits, in total 512B memory.

You may have noticed this line in the RAM logic: address_in(7 downto 0).

By choosing one more bit (making $2^8 = 256$ locations), it ensures flexibility and possible scalability for the future if needed. Using more bits prepares the design for addressing more locations in the future without changing the code.

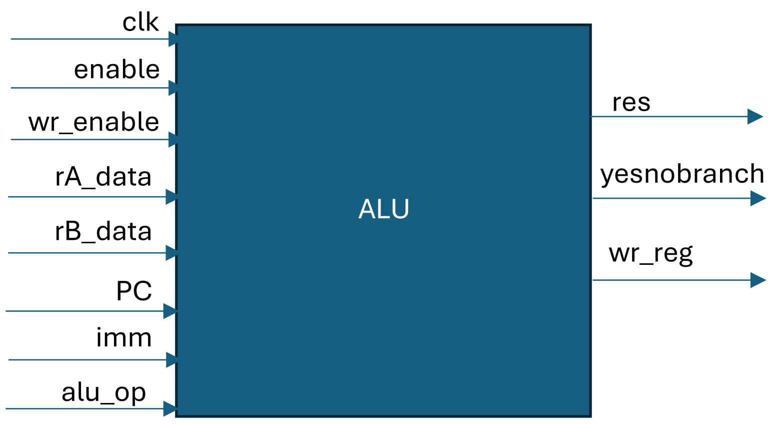
Part 3

The ALU (Arithmetic Logic Unit) is the next component we need to implement. It takes the opcode given by the decoder, along with input data read from register file, and output a result that can be written to the register file.

Note that in this CPU, the ALU will only do single cycle operations. Whilst a whole instruction will take multiple cycles to complete, the ALU itself will output a result in a single clock cycle (to keep things simple). The other cycles are due to decoding, fetching and any writebacks of results which are required.

In this ALU, I will code a case statement and assign to an internal register the operation of choice, before then writing the contents of that register to the output ports.

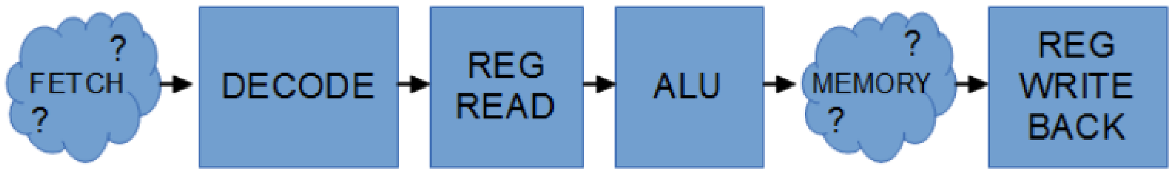
Note – when doing add/sub operations, there is a chance of overflow. Because of that, we need to make sure to handle overflow correctly. Also, the load and store operations are identical. The ALU itself lacks access to RAM or register file. To perform actual load/store operations that interact with memory, we will need a top-level control unit. This control unit would interpret these ALU signals to enable or disable RAM reads/writes and route the data accordingly.



<u>Part 4</u>

After creating the modules in the previous chapters, we need to create a unit that is responsible for telling other units what to do, and when. For that reason, we need a control unit.

Let’s look at how our pipeline works:



To keep things simple, we don’t execute the next instruction until the current one has traversed every stage. Since all the units we’ve created so far have enable ports, we can get a control unit to synchronize everything up and drive those enable bits. The control unit will have one output, a bitmask showing the pipeline state currently active, as well as a reset and clock input. Each clock cycle the state will increment by one bit, and if reset is high it will reset to the initial state. The control unit is basically a state machine.

We have a “special” case when getting to the ALU – when our ALU needs to load/store information.

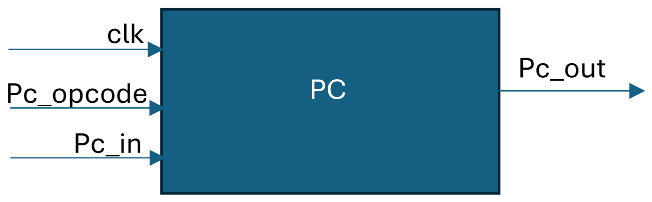
Part 5

The PC (Program Counter) unit holds the address of the next instruction to be executed. After each instruction, the PC is incremented to point to the next instruction. If there's a branch (e.g., a jump instruction), the PC will instead be updated to the target address specified by the branch.

Our PC will have an input for setting the next PC value, and the ability to stop/stay at the same location. We need the ability to stop at the same location due to our pipeline being several cycles long. With different possibilities for the PC unit operating, we can have an input to it which dictates the operation on the PC to perform:

- Increment PC
- Set PC to new value
- Do nothing (halt)
- Set PC to our reset vector – 0x0000

Our PC should look like this:



Part 6

Now, we are at the final part. We must connect all our components together and test them out.
To do so, I will create a final entity – the top-level module called “cpu” and import all the components into that module. There I will also “solve” the load/store mechanism that I have previously discussed in the ALU section. I will also create a testbench to simulate different scenarios to ensure that my design is working as intended.