Enkel

Anaykumar Joshi
anaykumar.joshi@gmail.com

July 13, 2013

Abstract

Enkel is a minimalistic, educational 8-bit softcore processor written in Verilog HDL. Enkel was the result of a course project, under the course EE-224. being taught by **Prof.Sachin Patkar** at IIT Bombay in 2011**.** This document has been written to detail down the structure of this minimalistic processor. Hopefully, this would serve as a first step for anyone trying to understand the internals of processors, in a very hands-on manner.

1 Overview

Enkel is based on Von Neumann architecture (less hardware compared to Harvard architecture). Eight fundamental opcodes are visible to the user:

1. **ADD** xxxx0
2. **NOT** xxxxx
3. **GET** mem
4. **PUT** mem
5. **FEED** constant
6. **LD** xxxxx
7. **JMPB** xxxxx
8. **SHOW** mem

Every instruction is a 8 bit string; with the first 3 bits encoding the operation. Example, ADD :: 000, NOT :: 001, GET :: 010 ... SHOW :: 111. 'x' stands for a don't care bit. The last instruction of the program should always be 0x01 (i.e ADD xxxx1). This word indicates end of program. The program execution is stopped by CPU by enabling master reset which puts the system back in idle state. Our computer contains 2 special purpose register : A and B . All the major arithmetic and data transfer operations in the computer revolve around these two registers.

2 Brief Explanation of Opcodes

1. **GET** mem : Takes the value stored in address pointed by mem, and stores the data in a special purpose register named B.
2. **PUT** mem : Takes the value from A and stores the data in memory byte addressed by 'mem'
3. **ADD** xxxx0: Adds the numbers stored in A and B and stores the result in A.
4. **NOT** xxxx1: Stores the bit-wise compliment of value stored in B into register A.
5. **FEED** constant : It stores the value "constant" into B.
6. **LD** xxxxx : Adds the value of Program Counter to the value stored in B, and stores the result in PC.
7. **JMPB** xxxxx: While doing computational operations (e.g. Addition, NOT,etc.) a special purpose bit (carry bit) would be present in the ALU which would contain the carry information with respect to the latest arithmetic operation. JMPB checks the value of this bit . If it is 0, JMPB does nothing. Else it performs the function of LD
8. **SHOW** mem: Feeds the value stored in the mem location to a register which is directly joined to the output LEDs. It basically is the only way our computer can talk to the outside world

Data type: The Data can only be in the range -127 to +127 As of now, the memory in RAM which can be accessed for storing and accessing data is limited to 32 addresses (this is because of the length of IR). This limitation can be removed by increasing the size of IR from 8 to 11.

The RAM is logically partitioned into 2 parts. The upper part starting from 0x00 stores the program. The lower part starting from 0xF0 stores the data. Here is a list of building blocks used in the construction:

1. **increment.v** : used in ALU for incrementing PC
2. **register.v** : used in REST and ALU for implementing A,B,PC,IR,MAR and latching registers.
3. **adder.v** : used in ALU
4. **compliment.v**: used in ALU
5. **tri_state_buffer.v** : used in REST for interfacing with RAM
6. **d_ff.v** : used as a building block for d_array
7. **d_array.v** : used in CPU for implementing DFF chains
8. **3to8dec.v** : used in CPU to decode opcode
9. **mux.v** : used in COMPUTER's testbench to switch from programming mode to execution mode
10. **carry_register.v** : used to store the value of carry bit
11. **choose.v**: has 2 inputs ([7:0] wires) , select pin , enable pin. output is one of the inputs depending upon select line. if enable is low, output is 8'bzzzzzzzz

These blocks are used to build ALU.v , CPU.v , REST.v , RAM.v These 4 modules are invoked in COMPUTER.v

3 Description of different modules:

The overall input to the computer as a black box would be a clock, and a start input. The REST, RAM and ALU block are capable of exchanging information between them. CPU triggers the interaction between various modules depending upon the op-code

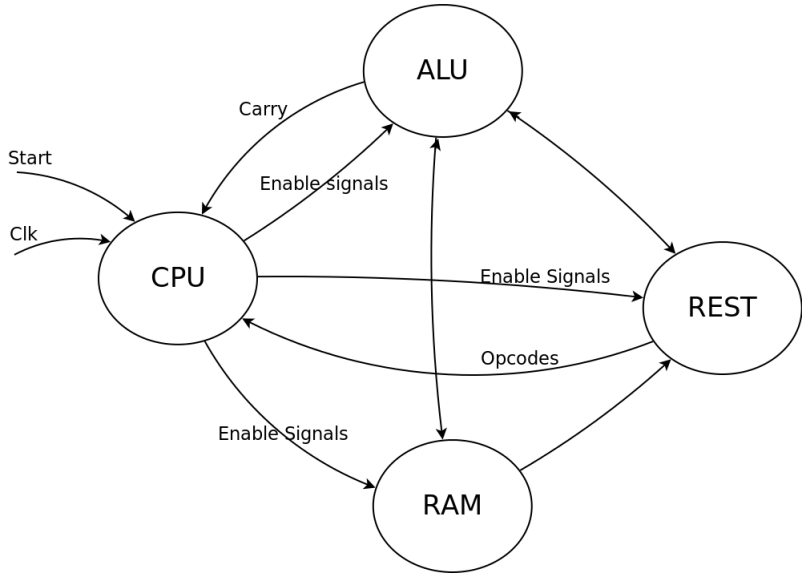

Figure 1: Bird's Eye view of Enkel

3.1 Arithmetic Logic Unit (ALU)

The Arithmetic Logic unit that we have implemented consists of two general purpose registers, A and B. All the arithmetic operations in the computer revolve around these 2 registers. The input to the ALU are the enabling signals from CPU, data value from RAM, value of PC. Output from ALU are data value to RAM, revised value of PC, and the special carry bit. The modules "ADDER" and "COMPLEMENTER" are the prime blocks of our ALU. The adder adds the numbers stored in register A and register B, and stores the result in register A, while the complementer block takes the bit-wise complement of the number stored in B and stores the result in A. In addition, ALU contains some latching registers which perform the function of locking the revised values of A / PC which are to be loaded in A / PC. We introduced these latching registers when we encountered some problems of feedback from the adder and complimenter. i.e the result of adder/complimenter was stored in A and the new value of A was again being added with the value stored in B and this process kept on changing the value of A. Also, there are a few blocks which we have named "CHOOSE". These blocks are basically a series of multiplexers. They decide (on the basis of signals from CPU) whether the operation performed involves A or PC and adder or complimenter

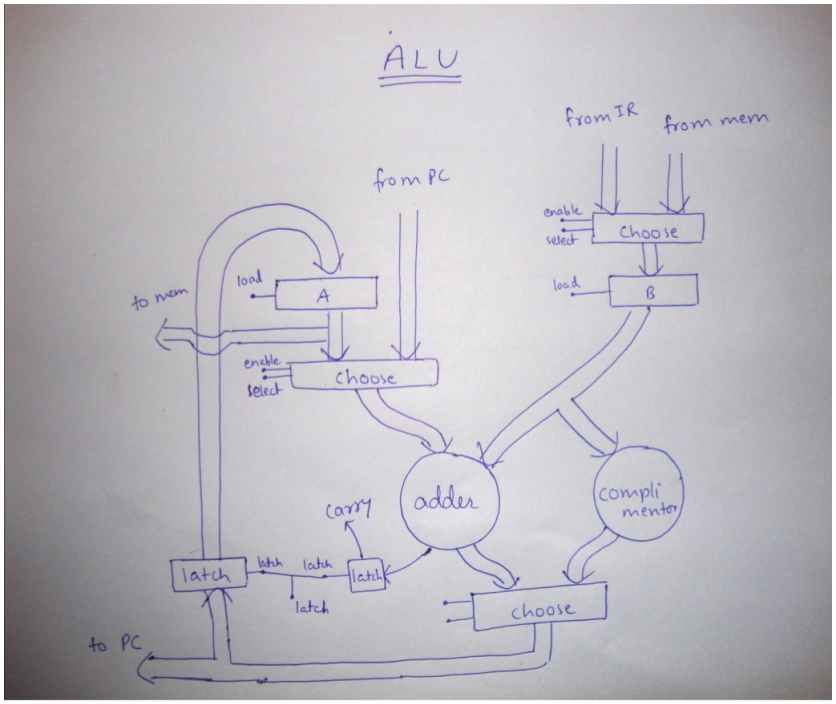

Figure 2: ALU Architecture

3.2 CPU Architecture

The CPU consists of (in all) 10 chains of flip flops. Each chain is of length 5, i.e each chain consists of 5 flip flops attached back to back. The processor starts when a pulse is triggered at the input terminal (this pulse encounters just one falling edge of the clock). As soon as the CPU receives the START signal, the first thing that it does is to reset all the registers and flip flops in the CPU, as well as send signals to ALU and the REST of the circuit to reset all the components. This process is called MASTER RESET, and is carried out only once when the computer starts. Now, the system is ready to fetch the first instruction. The user is expected to have put the code starting from address 0x00. Our CPU is based on "a rippling ONE", i.e. throughout the process of execution of the code, a single '1' ripples through the Flip-Flops in the CPU. After the start & reset process, the fetching process begins. The fetching process picks up the value (from RAM) corresponding to the address in the Program Counter. This process requires the first chain of 5-DFFs. Fetch process is followed by a process that we name "Next". The name is so given, because this stage prepares the CPU for executing the next line of code by incrementing the value of PC by 1. (Next process could well be considered a part of Fetching, but we preferred to give it a separate name) . . After the Fetch process, Instruction register (IR) is loaded with the value drawn from RAM. The first 3 bits of IR are sent to the CPU, (these bits form the OPCODES of our computer).Note that, the "Next" process has no effect in the execution of the current piece of code. It simply increments PC for executing the next piece of code after the present piece is completed. The opcode is sent to a decoder present inside CPU. Depending upon the opcode (which is the first three bits of the number fetched from the RAM), one of the 8 output Pins of the decoder is pulled high. The rippling 1 then enters one of the 8 chains corresponding to the 8 opcodes, namely ADD, NOT, GET, PUT, FEED, LD, JMPB, and SHOW. Each of these stages are again chains of DFFs. Lets focus on one of these

chains, say ADD. There are 5 DFFs in ADD attached back to back (actually there are 5 FFs in every chain). The outputs, Qi s, of these FFs are sent to a combinational block. The output of the combinational block is a number of enabling pins, which the CPU is supposed to send to the various units of the computer (example, adder, RAM, MAR, PC, etc.) Now, each of the 8 chains produce such enabling signals for the other blocks of the circuit. These enabling signals (corresponding to different chains) are further sent to a master combinational block; the output of this block is the final output of the CPU. The output of the 5th DFF of all the chains are inputted to an OR gate, and the output is given to the fetch cycle via an OR gate. Due to this, once an instruction has been carried out (which means the rippling "1" has reached the fifth DFF's output), a signal is sent to the Fetch chain, to initiate a new Fetch and Next process. This cycle goes on and on until the IR receives a special value from the RAM which indicates end of program. The CPU also has a few inputs. for example, a special bit which keeps track of the carry from the latest arithmetic operation. This bit helps us to use the OP-Code of JMPB, whose working is explained earlier.

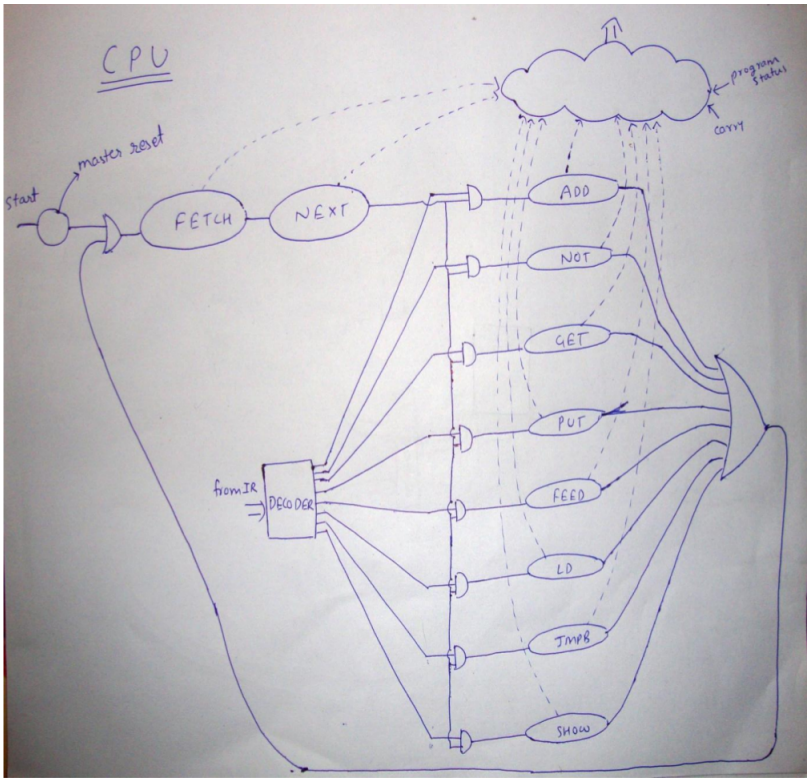

Figure 3: CPU Architecture

3.3 Rest Block

The name given to this block is REST because it performs functions which are not easy to classify under one name. So we named it REST (as in..rest of the circuit). This block mainly contains Instruction register, Program counter, Memory Address register (MAR), incrementer. This is the place where the status of the executing code is present. It performs critical operations like passing op-code to CPU and keeping track of which instructions to fetch. It does

these steps with the help of Program counter, Memory Address Register(MAR) and Instruction Register. This unit also has ports 'Address' and 'Data' for interfacing with RAM. Along with them , there are many "choose blocks" which are basically a series of MUXes. These blocks perform the function of choosing the appropriate unit under the control of CPU.

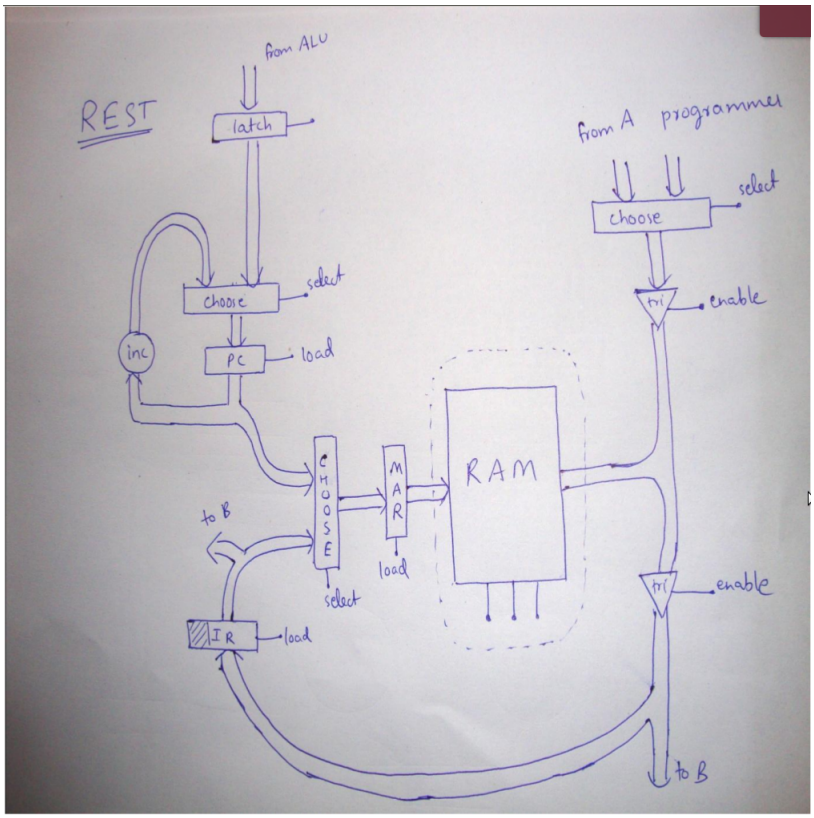

Figure 4: Rest of the System

4 A Simple Test Case

To test the computer using a FPGA, we put a code inside the RAM and started the computer and observed the signal and data flow. Here is one of the codes which we tested:

Address	Instruction	Functioning of the instructions
00000000	01011100	GET data from address 11100000
00000001	00100000	NOT data in B and store it in A
00000010	01011101	GET data from address 11101000
00000011	00000000	ADD it to data stored in A and sore result in A.
00000100	10000011	FEED a constant value 3 in B. (for implementing JMPB)
00000101	11000000	JMPB conditional jump. (if condition)
00000110	11111110	SHOW the data stored in RAM corresponding to address 11110000
00000100	10000001	FEED a constant value 1 in B. (for implementing LD)
00000111	10100000	LD jump (else condition)
00001000	11111111	SHOW the data stored in RAM corresponding to address 11111000
00001001	00000001	end the program

The data to operated on was also stored in RAM.

Address	Data
11100000	10011010
11101000	00000001
11110000	00000010
11101000	00000001

The above mentioned code compares 2 numbers stored in memory addresses 11100000 and 11101000 and if the second number is greater than the first number, it displays the number stored in 11111000 as output; else it displays the number stored in 11110000 as output.

4.1 How this code works

If the second number is greater than the first number, the subtraction process carried in the first 4 instructions produces a carry bit in the ALU. This carry bit an input to CPU. When the CPU executes JMPB instruction, it checks the status of this bit. Suppose this bit is high, CPU would jump by a value 3 (because a constant 3 was fed into B in the previous instruction) and SHOW the number stored in 11111000 and then the program ends. Suppose this bit is low, CPU would ignore the JMPB instruction and would execute the SHOW instruction to output the number stored in 11110000 and then LD instruction takes the program flow directly to the end instruction and the program ends.

5 Challenges

- **CPU Architecture:**

Due to the sheer number of wires and connections, it was difficult to build the combinational logic blocks. Even when we had confidently built it, some small errors came up which were pretty difficult to remove. This was overcome by keeping logical names of wires and building blocks of the system, to simplify the process of back tracking.

- **Interfacing**

While building CPU, we needed to build the logic for the combinational blocks. This required a very accurate note of the enabling signals. For this, we needed to build a huge testbench to finalize the values of the sequence of enabling signals which worked. This was especially tough because, even after constructing the CPU, the only way to verify that we had done it correctly was to complete the construction of the entire computer, and then burn some code in the RAM in testbench. While running the testbench for the entire computer, we came across a few logical errors which we had not thought about before constructing CPU. For example, The chain of GET, and FETCH tried to access RAM simultaneously; an error which took a pretty long time to identify. The error occurred because the combinational logic in CPU was also a function of decoder output, and the decoder output was stabilized just after the FETCH cycle. Thus, after the GET cycle, while executing the next opcode, there were unexpected output levels. We eventually backtraced and found out the error. Also, ALU and REST required big testbenches to check all the functionalities they provide.

- **ALU**

Since the ALU was the first major block that we had built, we had not exactly decided its interface with the remainder of the computer. Thus, we needed to modify the ALU a few times to make it compatible with the remaining computer.

- **RAM Interfacing and Implementation**

Implementing RAM and interfacing it with the rest of the circuit was probably the toughest link in the project. This was because we were, initially, unaware of tri state buffers. We faltered numerous times while trying to cope up with RAM. To check the working of the computer, we needed to burn some code inside it. Initially, we had built a separate code for programming the RAM. But, to test the computer, we needed a programmed RAM in the testbench of Computer. This was not possible if the RAM was to be programmed using a separate code. Thus, we added a functionality of programming the RAM from within the Computer. While programming, the address lines of the RAM are selected from outside the Computer. But the words to be fed inside the computer was fed as input to Computer. This was done because we couldn't find a way of putting data externally and then making this very programmed RAM available for the Computer to work on. The DATA variable had to be used as an INOUT port as it was both giving outputs and taking inputs through the ports.

- **Carry Bit**

One tricky part in the construction of CPU was the JMPB block. If carry bit is high, JMPB is supposed to add the value of PC with the present value in B and store the result in PC; if carry bit is low, JMPB is supposed to be ignored and the next instruction should be executed. There were 2 logical errors in our initial design of this part. 1. Carry bit used to have undefined logic level whenever JMPB needed it. This was because we had not latched the value of carry. We added a latching circuit in the ALU and this part was solved. 2. The rippling 1 used to disappear if carry was low. This was because, our initial condition for starting JMPB chain was the condition that carry should be high. This worked if carry was 1, but for carry = 0, the rippling one vanished and the computer effectively stopped. This was overcome by removing the dependence of starting the JMPB block on carry. Instead, carry was made a part of the combinational logic block.