16Bit microprocessor with 8 registers, its implementation in verilog & vivado simulation

Bavly Kirolos
TUM BIE Campus Heilbronn
Bavly.kirolos@tum.de

I. INTRODUCTION (*HEADING 1*)

The development of microprocessors has revolutionized computing and embedded systems, especially as the need for more specialized and efficient components arises. Educational environments often require simple, understandable processors to teach fundamental concepts. This processor is a 16-bit processor designed to fulfill this need, offering a balance between simplicity and functionality. This processor in word addressable rather than byte addressable and all registers and wires mentioned in this paper are 16 bits by default unless specified otherwise.

This paper details the design and implementation of Processor, covering its architecture, instruction set, memory management, and performance characteristics. Please keep in mind that the main focus for this project was the functionality and readability of the code rather than efficiency. A more efficient model offering the same functionality might be available in the future.

II. PROCESSOR DESIGN ARCHITECTURE OVERVIEW

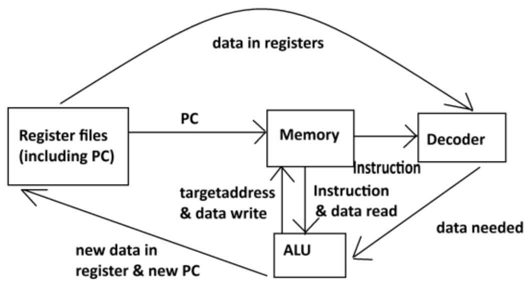
A. *Processor*

This is the Top module, it includes the Program Counter register which is used to store the address of the instruction currently being executed, 8 registers for programmers to use and the needed wires to connect these registers to 3 main components: memory, decoder and ALU (which includes a full adder), and to also to connect these components with one another.

These registers are initialized to zero and are continuously updated with the rising edge of the clock. The process starts with the Program Counter register, it is connected as input to the memory. The memory outputs the instruction needed to the decoder and the ALU. The decoder uses this instruction and the contents of the 8 registers as input and in return it outputs the actual data meant as input for the ALU.

This data includes the data in chosen ra, rb, rc registers and a signed immediate, some of these data are redundant depending on the instruction coming from memory. It is the ALU’s job to find which should be used in calculations and which to ignore. The ALU also gets input from memory, current PC form the PC register and also clk as input. The ALU and memory are connected by 3 wires: datatobewritteninmem, datafrommem and targetaddr, this allows the ALU to write data into the memory and to request data from a specific address.

The memory sends this data in the same cycle so the ALU can use it in its output. To the registers. The ALU is responsible for updating the PC and updating the specified output register based on the instruction it received as input and the state of the registers before execution


data in registers
Register files (including PC)
PC
Memory
Decoder
Instruction
targetaddress & data write
Instruction & data read
data needed
ALU
new data in register & new PC

Simplified representation of the components in processor

B. *Memmory5 component*

The memory is initially initialized with zeros and then populated with the assembly code written specifically for this isa.

```
integer i;
  initial begin
  for (i=0; i<SIZE; i=i+1) begin
    MEM[i] = 'h0;
  End
if (MEM_FILE!=0)      $readmemb(MEM_FILE,MEM);
```

The memory receives input from PC register we mentioned earlier whenever it gets updated, this means that the processor in finished with the previous instruction and is requesting a new one(or is just starting and requesting instruction at address 0), the memory outputs the instruction inside the requested address and passes it to the decoder and the ALU units, The memory also receives a 16bit input from the ALU, the first 2 bits specify if the ALU wants to read or write data and the remaining 14 bits are the address where data should be written or read from, the data read output register is updated whenever the read bit is triggered, this insures that the ALU gets the data needed as fast as possible.

```
always @(*) begin
if(dataaddr[15]) dataread= MEM[dataaddr[13:0]];
```

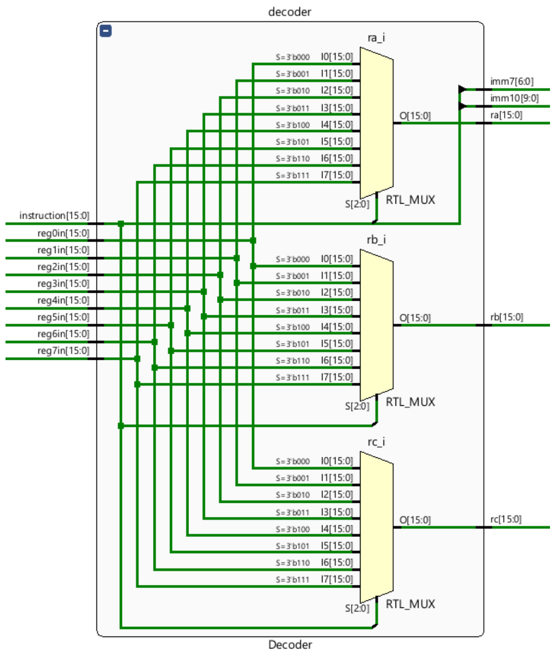
The memory keeps outputting the data even when it is redundant as it is the ALU’s responsibility to check if it needs this data or if it should be ignored. When writing data to the memory however, the memory registers are updated at the clk posedge if the datawrite flafgis on, this synchronizes the memory with the processor.

```
always @(posedge clk) begin
if(dataaddr[14]) MEM[dataaddr[13:0]] = datawrite;
```

This ensures that only the specified address is changed during one cycle, if it were to use the same mechanism as it does for reading, the memory would write in the intended location but also at address 0 and the last address stored in the sum register which is unintended. This was initially a bug in the earlier versions of memory because it was not obvious until the components were connected and the assembly program was being tested, more details on that part in the testing and debugging section.

C. *Decoder component*

The decoder gets the instruction from the memory and the data in the main PC register and 8 programmers' registers. Instructions specify 1, 2 or 3 registers needed as input and some instructions include input themselves, in each case the decoder outputs the data needed in the specified register along with other data that might not be useful but that is not a problem, as long as the needed data is output in its designated wires. This is realized by internal multiplexers with data from the registers as data input and specific bits of the instruction as the select input, this component has in total 3 multiplexers outputting 3 outputs and 2 outputs are [6:0] imm7 and [9:0] imm10 which are the last 7 and 10 bits of the instruction unchanged.



D. *ALU component*

The ALU is the most critical component here, this is because it is the module responsible for updating the registers and the memory and thus it is what changes the "state" of the machine. It does that in 2 consecutive steps, calculating the new data that should be stored and choosing which register to store this new data. It contains a full adder which takes operand1 and operand 2 registers as input and outputs the sum, the mechanism of how it works will be discussed in the next module. ALU uses imm7 to calculate imm7extended

```
wire [15:0] imm7extended = {{9{imm7[6]}}, imm7};
```

and immreadytosub

```
wire [15:0] immreadytosub = ~imm7extended+1;
```

imm7 is now ready to be used in addition and subtraction as it can be stored in operand1 and 2 registers. Based on the instruction (most 3 significant bits of it) the ALU stores imm7extended, immreadytosub or any of the inputs from the decoder as operand1 and operand2 using multiplexers.

- For ADD, ADDI and SUBI the sum is stored in racalculated register which is the register that stores the new data we want to write in one of the processor registers
- For BEQ the sum is stored in PCout instead as this instruction calculates the new PC if the 2 input registers are equal, this allows programmers to jump in memory locations (subroutines) based on if conditions.
- JALR doesn't use the full adder at all, it simply stores the next pc in racalculated (so it can be later stored in a specific register) and stores rb in PCout. This is used to call a subroutine and store the returning address in a specific register. The return address can't be hardcoded in the assembly program as the subroutine can be called a few times from different callers.
- LUI simply stores the given imm10 followed by 6 zeros in the spicified register.
- SW sends the specified input register to the register dataouttomemory which the memory takes to store in a specified location. This location is also calculated using the full adder and the sum is stored in the lowest 14 bits of targetaddrinmem , the first 2 bits are 01 indicating that we don't want to read from memory, instead we want to write data in the address specified by these 14 bits. We only have 1024words in the memory so we shouldn't need more than 10 bits for the address anyway.
- LW uses the same mechanism to calculate the address however the first 2 bits of targetaddrinmem are 10 indicating we want to read data but not write. The datainfrommem is then stored in racalculated so it can be used later to update the actual register that we want updated.

Racalculated is now ready but not yet stored in any of our registers, notice that all instruction except for BEQ and SW update one of the registers. That's why we update the registers only if neither of these instructions were executed

```
if(instr[15:13] != 3'b011 && instr[15:13] != 3'b110 )
```

We then choose the register to store racalculated in based on ra specified in the instruction .... inst[12:10]

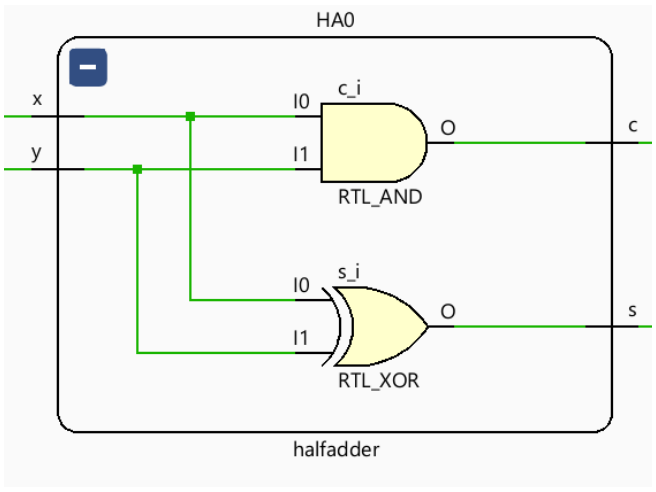
E. *Full adder*

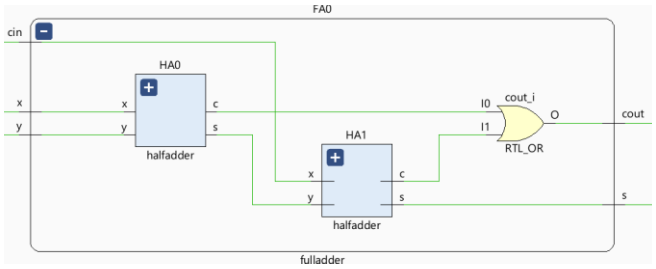
Inspired by Sameh Nour's Verilog Primer, This module is a helper module for ALU, it accepts 2 inputs from it, and outputs their sum. To explain this module lets starts with the half adder

```
assign s = x ^ y;
assign c = x & y;
```

This uses 2 input bits to calculate sum and carry. If we want to add more than 2 bits, we must assume there might be a carry-on bit from the previous 2. This brings us to the full adder with 3 input bits (operand 1 & 2 & carry on) and 2 output bits: sum and carry-on.

This is done by taking the sum output as an operand in another half adder and carrybit as the other operand. The sum of the second half adder is the real sum and the carryout output is triggered if carry out of either one of the half adders is triggered. Connect 16 full adders together each carry out trigger the carry in of the next and you will get a 16bit full adder.





III. Testing and Debugging

This project was developed using continuous testing and integration methods, meaning that each component was tested right after being written and for big components a few intermediate chokepoints were tested.

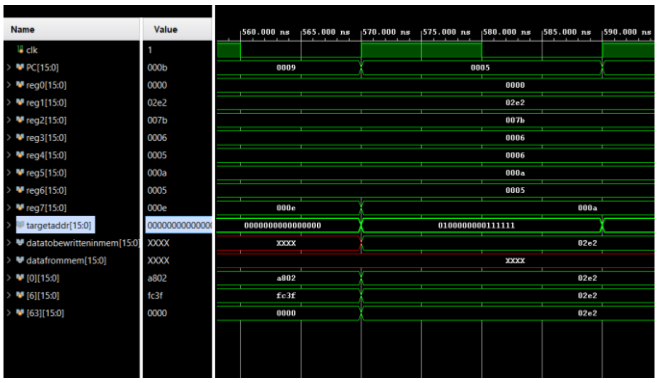
For example, the first version of memory was ROM that had only one input: address and one output: instruction, it would simply output the instruction stored in the requested memory address. Writing to memory in the earlier versions was written like this : always @(*) begin

```
        if(dataaddr[14])

        MEM[dataaddr[13:0]] = datawrite;

        End
```

Which is the same way as the read instructions, and it worked fine when the processor was executing the write instruction as the first instruction. When testing this instruction iside a long assembly program however , this was the result



The memory succeeded in writing the data in the required address as expected, but it also overwritten the data at address 0 and address 6 for no apparent reason, upon further testing it was found that the addresses are always 0 and whatever was in sum register during last clock cycle.

This makes sense because new instruction input to the alu would turn on the write flag once it is receive before the sum would be properly calculated. In short Propagation delay was not taken into account.

The new memory had a clk input to make sure it is synchronous with the processor and that the memory will only be updated after all registers have been correctly calculated and ready to be transmitted to the memory.

```
    always @(posedge clk) begin

        if(dataaddr[14])

        MEM[dataaddr[13:0]] = datawrite;

    end
```

Now lets test the following assembly program

//this assembly program multiplies 123 by 6 then stores the result in memory and then reads it back again

//r0 = 0

//r1 is the result

//r2 is the number we add each time . multiplicand

//r3 is the number of times we add it . multiplier

//r4 is counter

//r5 is start address of the subroutine

//r6 is the return address to the main program

//r7 is trash and then overwritten with the result as read from memory.

LUI r2, 2 // this stores 10 followed by 6 zeros , ie stores 128 in r2

SUBI r2, r2, 5 // substract 5 from 128 . r2 expected = 123

ADDI r3, r0, 6 // add 6 to 0 and store it in r3

ADDI r5, r0, 10 // add 10 to 0 and store it in r5

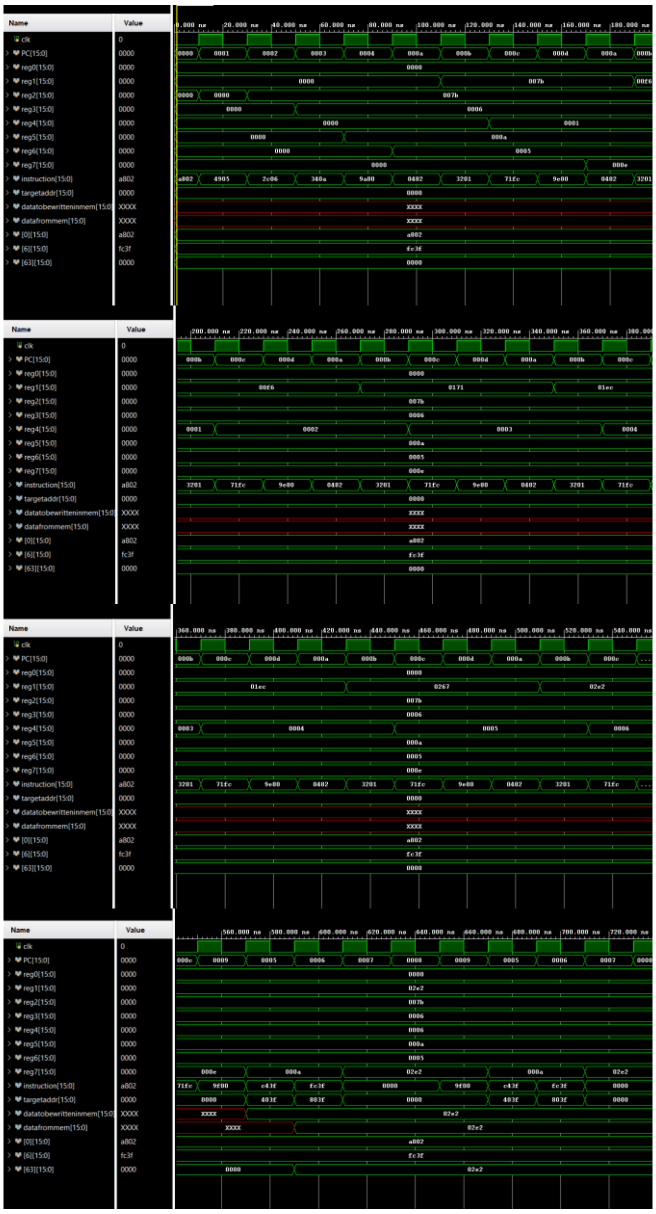
JALR r6, r5 // store return address in r6 and jump pc to address in r5

SW r1, r0, 63 // strore result in mem@63

```
lW    r7, r0,63     // store content of mem63 in r7 //we should
expect our result
ADD  r0, r0, r0     // store 0+0 in r0
ADD  r0, r0, r0     // store 0+0 in r0
 JALR r7, r6        // jump back to the main program
                 // sub call calls here (address stored in r5)
add  r1, r1, r2  // increment r1 by r2 , r2 = 6
addi r4, r4, 1   //increment counter (r3) by one
BEQ  r4, r3, -4  //go 5 steps back so we can exit the loop
jalr  r7, r5     // loop again
```

Another program was also used to test the processor but it is less intuitive as it tests all instructions one after another but there is no useful info calculated by this program :

```
LW   r1, r0, 0             // store mem@ 0 in r1 (0)
LW   r2, r0, 16            // store mem@ 16 in r2 (1)
SW   r2, r0, 17            // store r2 in mem @17 (2)
LUI  r3, 111_111_1111       // load r3 (3)
 BEQ  r4, r1, 2            // should be false (4) if true , skip 2 pc
BEQ  r4, r5, 1          //  if true , skip 1 pc
LUI  r5, 111_111_1111       // should be ignored (6)
JALR r6, r2                // jump to pc11 , store 7 in r6 (7)
 LUI  r5, 111_111_1111       // should be ignored (8)
LUI  r5, 111_111_1111       // should be ignored (9)
LUI  r5, 111_111_1111       // should be ignored (10)
SUBI r7, r2, 9           // store 11-9 in r7 (11)
ADDI r6, r6, 3             // add 3 to r6 (12)
 ADD  r7, r6, r7             // add r6 to r7 (13)
SW   r1, r0, 15 // store 1110010000000000 im mem 15 (14)
0000000000000              // (15)
0000000000010                    // (16)
```



IV. Conclusion

This project is for educational purposes as it shows on an abstract level how a processor works, and which components are necessary such as instruction decoding, ALU operations, memory read/write mechanisms, and register updates.

It is a simple yet functional structure. Understanding such concept for me is the main takeaway from the HDL course, I can finally say that the gap between physical transistors (that i studied in engineering) and assembly that we learned in the first semester has finally been closed. It also helped in learning a lot of Verilog syntax itself and what could be written to synthesize various parts.

If I were to implement this project again, I would not have done much differently than this , because the project evolved a lot while I was working on it, and I changed various parts a few times. I am sure it would take less time and effort though because of what I learned from this one.