IPA Project Report

Team ATmicro

Objective

To implement a Y86-64 instruction set based processor using Verilog HDL in 2 ways

1. Sequential implementation
2. 5-stage Pipelined implementation

Instructions supported

- halt
- nop
- cmovXX
- irmovq
- rmmovq
- mrmovq
- OPq
- jXX
- call
- ret
- pushq
- popq

ALU Design

Approach for the ALU

The ALU consists of 4 main 64-bit operations

- Add
- Subtract
- AND
- XOR

We will now discuss the logic for each of them one by one.

ADD_64

- ADD_1 : We designed a 1 bit full adder, ADD_1, that takes 1 bit input of three registers and returns the sum and carry.
- We then instantiated the ADD_1 module inside the ADD_64 module for each bit (64 times).
- We store the resulting sum and carry in a 64 bit signed register

SUB_64

- - NOT_64 : We designed 64 bit NOT gate, that takes 64 bit input and returns the complement of the number by taking NOT of each bit
- We instantiated the NOT_64 inside SUB_64 to get the 1s complement of the second operand.
- We used ADD_64 inside SUB_64, to add 1 to the complement to get the 2s complement.
- The two's complement is added to the first operand using ADD_64.

XOR_64

- We iterated through each bit and called the inbuilt xor function to calculate the xor of the input bits.

AND_64

- We iterated through each bit and called the inbuilt and function to calculate the and of the input bits.

Control Input to ALU

We defined a parameter "ctrl" in wrapper_test.v, which determines the kind of operation to be carried out.

- ctrl = 0 → Addition
- ctrl = 1 → Subtraction
- ctrl = 2 → AND Operation
- ctrl = 3 → XOR Operation

Sequential Implementation

The Basic Working Principle

The basic working of the sequential design can be summarized as

1. On the positive edge of the clock cycle, a new instruction is fetched from the instruction memory
2. Each block processes the instruction and obtains a set of outputs which are then sent to the subsequent blocks.

This keeps on happening as long as the instruction memory is not depleted or the processor does not run into faults like bad addressing or halt instructions.

1. Fetch block

- It contains a 1 KB instruction memory block.
- The fetch block takes 10 bytes of data from the instruction memory and builds a 64 bit instruction.
- The icode, ifun, rA, rB, valC and valP values are then extracted from this instruction and given as outputs to the later stages of the processor.
- At each positive edge of the clock, a new instruction is generated based on the value of the PC.
- This new instruction is then used to obtain the required output variables by splitting it into appropriate lengths.
- This block also outputs 2 status variables namely instruction valid (instr_valid) and memory error (imem_error) which denote the proper/improper working of the fetch stage.

2. Decode and Writeback Block

- It contains an array of 15 registers which form the register memory. (out of these 15 registers we have specified the 4th one (%rsp) as the stack pointer register)
- At each positive clock edge, the decode block takes icode, rA and rB and accesses the register memory using these values in order to generate

actual numerical values valA and valB and store them in registers for use in successive stages.

- At each negative clock edge, the writeback block takes valE (output of execute block), valM (output of memory block) and icode and updates the values stored in the register memory using the input values.
- In case of call, ret, pushq and popq instructions, the writeback block updates the stack pointer to point to another location based on the input values.
- We have implemented Decode and Writeback blocks in the same Verilog module because both of them share a common register memory and hence can access it easily if they are defined inside the same module.

3. Execute Block

- It contains the 64 bit ALU which was coded earlier.
- It takes icode, ifun, valA and valB as inputs and returns valE as the result of the requested operation.
- The execute block also evaluates 3 condition code flags namely ZF (zero flag), SF (sign flag) and OF (overflow flag) which are used to implement conditional instructions like conditional move and conditional jump.
- The execute block also combines the values stored in valB and valC in order to perform memory based operations.
- It also evaluates a variable 'jmpcnd' which specifies whether a jump is to be taken or not based on the conditional codes evaluated earlier.
- In case of call, ret, pushq and popq instructions, the execute block updates the value of valB by either incrementing/decrementing it by 8, hence giving the location of the new stack pointer.

4. Memory Block

- It contains a 8 KB data memory block.
- At each positive edge of the clock, it takes icode, valE and valA and based on their value, it either updates the data memory, or reads a value from the data memory and stores it in a variable valM which is used for successive stages.

5. PC Update block

- At each positive edge of the clock, this block takes values like icode, valP, valC, valM and jmpcnd, generated by the previous stages of the processor and uses them to update the value of the program counter.
- This step makes the processor ready for the next cycle of execution.

Instructions to Run the Sequential Processor

1. Change directory to Sequential.
2. To compile the code, type the below given instruction on the terminal

```
iverilog final_test.v
```

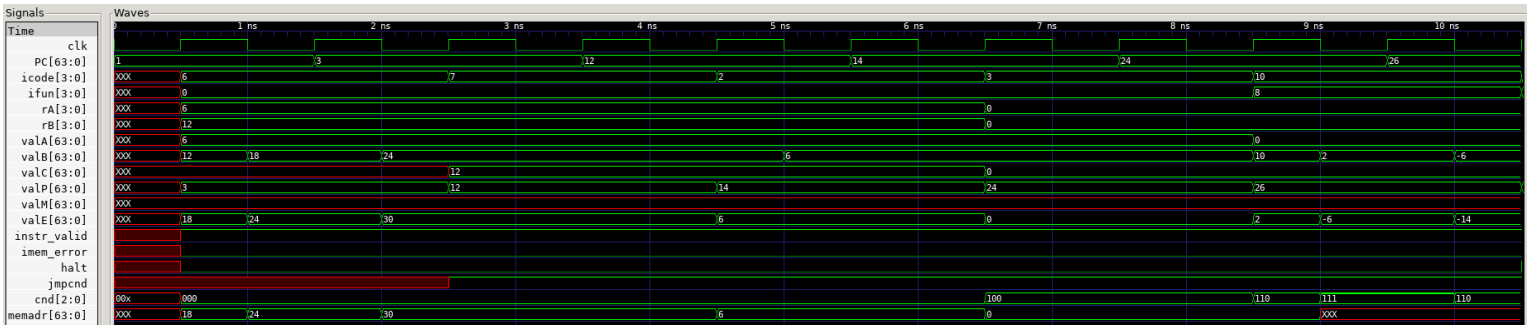
3. Run the executable generated using

```
./a.out
```

Outputs and GTKwaveform

```
tanmay@Tanmay-Dell:.../temp$ ./a.out
VCD info: dumpfile final_test.vcd opened for output.
```

clk	PC	icode	ifun	rA	rB	valA	valB	valC	valP	valM	valE	instr_valid	imem_error	halt	jmpcnd	cnd	memadr
0	1	x	x	x	x	x	x	x	x	x	x	x	x	00x	x		
1	1	6	0	6	12	6	12	x	3	x	18	1	0	0	x	000	18
0	1	6	0	6	12	6	18	x	3	x	24	1	0	0	x	000	24
1	3	6	0	6	12	6	18	x	3	x	24	1	0	0	x	000	24
0	3	6	0	6	12	6	24	x	3	x	30	1	0	0	x	000	30
1	3	7	0	6	12	6	24	12	12	x	30	1	0	0	1	000	30
0	3	7	0	6	12	6	24	12	12	x	30	1	0	0	1	000	30
1	12	7	0	6	12	6	24	12	12	x	30	1	0	0	1	000	30
0	12	7	0	6	12	6	24	12	12	x	30	1	0	0	1	000	30
1	12	2	0	6	12	6	24	12	14	x	6	1	0	0	1	000	6
0	12	2	0	6	12	6	6	12	14	x	6	1	0	0	1	000	6
1	14	2	0	6	12	6	6	12	14	x	6	1	0	0	1	000	6
0	14	2	0	6	12	6	6	12	14	x	6	1	0	0	1	000	6
1	14	3	0	0	0	6	6	0	24	x	0	1	0	0	1	100	0
0	14	3	0	0	0	6	6	0	24	x	0	1	0	0	1	100	0
1	24	3	0	0	0	6	6	0	24	x	0	1	0	0	1	100	0
0	24	3	0	0	0	6	6	0	24	x	0	1	0	0	1	100	0
1	24	10	8	0	0	0	10	0	26	x	2	1	0	0	1	110	0
0	24	10	8	0	0	0	2	0	26	x	-6	1	0	0	1	111	x
1	26	10	8	0	0	0	2	0	26	x	-6	1	0	0	1	111	x
0	26	10	8	0	0	0	-6	0	26	x	-14	1	0	0	1	110	x
1	26	0	10	0	0	0	-6	0	27	x	-14	1	0	1	1	110	x



Choosing the testbench

We have chosen the testbench for our sequential processor such that a majority of the operations and blocks get tested. To this end, we have manually constructed our instruction memory such that the processor passes through the following operations

1. OPq
2. jXX
3. cmovXX
4. irmovq
5. pushq
6. halt

We can see that the various processor variables like rA, rB, valA, valE, etc. get modified in accordance to the operations being performed.

Pipelined Implementation

The Basic Working Principle

The basic working of the pipelined design can be summarized as

1. On the positive edge of the clock cycle, a new instruction is fetched from the instruction memory
2. Each block processes the instruction and obtains a set of outputs which are then sent to the subsequent blocks.
3. The main point of difference from the sequential implementation is that each block works kind of independently here and hence multiple instructions can be present in different blocks at the same time.
4. This gives us an edge in performance over sequential implementation as there the processor has to wait for 1 instruction to pass through all the stages before fetching the next instruction, whereas in pipelined implementation, the next instruction can be fetched as soon as the previous instruction leaves the fetch stage.

This keeps on happening as long as the instruction memory is not depleted or the processor does not run into faults like bad addressing or halt instructions.

1. Fetch Block

- We combine the PC update stage present at the end of the sequential processor with the fetch pipeline register at the start of the fetch stage.
- This allows us to fetch the next instruction as soon as one leaves the fetch stage without waiting for the other stages to get completed.

PC Prediction

- We have also implemented the PC prediction method in this pipeline register which helps us predict the PC in advance before the instruction reaches the execute stage.
- The strategy for PC prediction is
    - Non-control transferring instructions : PC = valP
    - Unconditional jumps and call : PC = valC

    - Conditional jumps : PC = valC (60% accuracy typically)
    - Ret instruction : Don't try to predict
- In the latter 2 cases, there is a chance that the predicted value of PC might be wrong (depends on certain control variables evaluated in later stages of the pipeline). In such a case, we can obtain the correct PC as
    - Mispredicted jump : PC = m_valA
    - Ret instruction : PC = w_valM

2. Decode and Writeback Block

- The Decode and writeback stages are more or less the same as they were in the sequential implementation with the addition of decode and writeback pipeline registers which control the flow of data into these stages.

Data Forwarding

- We have also implemented the data forwarding technique in the decode stage.
- This means we do not need to wait for dependent instructions to pass through the writeback stage in order for a subsequent instruction to be decoded correctly.
- This is possible due to the presence of new feedback paths from the execute, memory, and writeback blocks which feed data directly to the decode block before the register file gets updated.
- This means that the processor can function without intermittent nop instructions, thus improving the overall performance.

3. Execute Block

- We implemented an execute pipeline register that is placed before the execute stage.
- The pipelined register is used to control the flow of data into the execute stage

4. Memory Block

- We implemented a memory pipeline register that is placed before the memory stage.
- The pipelined register is used to control the flow of data into the memory stage.

Instructions to Run the Pipelined Processor

1. Change directory to Pipelined.
2. To compile the code, type the below given instruction on the terminal

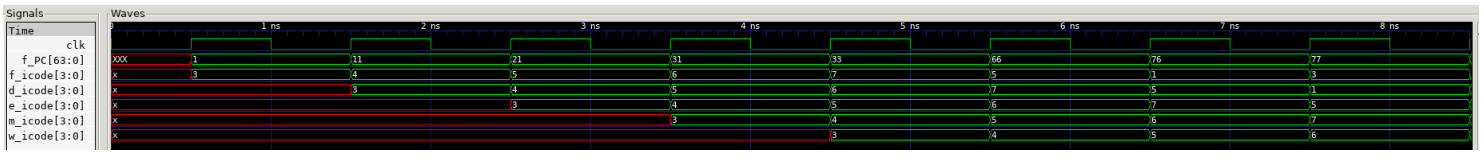
```
iverilog final_test_pipe.v
```

3. Run the executable generated using

```
./a.out
```

Outputs and GTKwaveform

```
tanmay@Tanmay-Dell:.../temp$ ./a.out
VCD info: dumpfile final_test_pipe.vcd opened for output.
CLK             f_PC            f_icode         d_icode         e_icode         m_icode         w_icode
0               x               x               x               x               x               x
1               1               3               x               x               x               x
0               1               3               x               x               x               x
1               11              4               3               x               x               x
0               11              4               3               x               x               x
1               21              5               4               3               x               x
0               21              5               4               3               x               x
1               31              6               5               4               3               x
0               31              6               5               4               3               x
1               33              7               6               5               4               3
0               33              7               6               5               4               3
1               66              5               7               6               5               4
0               66              5               7               6               5               4
1               76              1               5               7               6               5
0               76              1               5               7               6               5
1               77              3               1               5               7               6
0               77              3               1               5               7               6
1               87              0               3               1               5               7
```



Choosing the testbench

We have chosen the testbench for our sequential processor such that a majority of the operations and blocks get tested. To this end, we have manually constructed our instruction memory such that the processor passes through the following operations

1. irmovq
2. rmmovq
3. mrmovq
4. OPq
5. jXX
6. mrmovq
7. nop
8. irmovq
9. halt

We can also observe from the output that a new instruction enters each stage at positive clk edge. This shows that the pipelined construct which we wished to create has been established properly.

Challenges Faced

1. Adjusting the delays, such that various stages of the processor get the required inputs as and when they are required.
2. Defining instruction memory such that the maximum number of use cases can be verified.
3. Combining PC Update and Fetch Register stage in the pipelined implementation.
4. Checking conditions for data forwarding.

We overcame all the above-mentioned challenges and implemented the sequential and pipelined implementation of the Y86-64 Processor.