RISC Pipeline Project

Cameron Zach

Fall 2020

Architecture Description

The pipeline is composed of 5 stages – Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Writeback (WB). Each stage operates in two phases – on the rising edge of the clock, the stage writes its outputs to the next stage, and on the falling edge it reads its inputs into internal buffers for its next calculation. An overall block diagram for the pipeline appears in Figure 1. The designs of the five stages are as follows:

IF

The IF stage has just one output – a 22-bit instruction signal. This signal is taken from the ROM chip where instruction memory is stored, labelled IMEM on the block diagram. This ROM chip takes in a data address as input and outputs a 22-bit instruction. The address line is fed by the IF stage's internal Program Counter (PC) buffer, which increments by one on each clock cycle. To enable branching instructions, the IF stage also has a jump address input and jump input. When the jump input is high, the program counter is set to the value of the jump address, rather than incrementing by one. The block diagram of the IF stage appears in Figure 2.

ID

The ID stage is where the instruction is translated into operands for the other three stages. It takes in a single input, which is the 22-bit instruction. Its first step is to check the first four bits of the instruction, which indicate which instruction is to be executed. It then splits the remaining bits amongst its outputs and sets some other outputs as flags. The outputs are described in Figure 3. A block diagram of this stage is available in Figure 4.

EX

The EX stage is where all the calculations for the instruction occur. Before this can happen, the operands need to be fetched from registers. The EX stage is connected to the register file, which constantly outputs the values of each of the pipeline's four registers. The register file also handles forwarding, the details of which are discussed in the "Forwarding" section of this document. Each of the four registers are directed to their four respective inputs in the EX stage. When the `alu_imm_a` or `alu_imm_b` flags are set, the two input values `alu_val_a` and `alu_val_b` are fed directly into the ALU. When the flag is not set, the value is replaced with the value from the correct register, and that value is fed to the ALU in its place. The `mem_addr_in` input is also translated in this way, and passed to the `mem_addr` output. If the operation was a jump operation, then the `jump` and `jump_out` outputs are set to instruct the IF stage to jump to a new instruction, as detailed in the IF section. If the `ready_at` flag is set to 01, indicating that the value produced by the EX stage is available for forwarding, then the `fu_avail` flag is set to inform the register file that the current output of the EX stage should be used for forwarding. This flag, as well as all remaining outputs are all simply forwarded from their respective inputs for use in later stages. The diagram of the EX stage is available in Figure 5.

MEM

All access to data memory takes place in this stage. When the `needs_mem` flag is not set, this stage simply forwards its `data_in` input to its data_out output. When the flag is set, the stage performs an action based on the `mem_op` flag – it reads from memory if the flag is not set and writes to memory if it is. In the case of a read, data memory (labelled DMEM in the block diagram) takes in the memory address from the stage's inputs and provides a value to the stage's output. In the case of a write, the address is provided to DMEM, but the stage's data input is also provided to DMEM's `data_in` input, and the `write_enable` flag is set, causing the value stored at that address in DMEM to be set to the provided value. The new value is then passed to the stage's output. Regardless of any of these flags, the `fu_avail` flag is set when the `ready_at` input is set to either 10 (indicating that the value was available for forwarding as of the MEM stage) or 01 (indicating that the value was available for forwarding as of the EX stage). Again, this flag and all remaining inputs are passed to the next stage. The diagram for the MEM stage is available in Figure 6.

WB

The final stage of the pipeline is the Writeback stage, where the result value is stored into its destination register. The data in and register in are directly connected to the data out and register out. Then if the `ready_at` input is set to 00, then there is nothing left to do – the instruction does not store a value in any register. Otherwise, the `write_enable` flag is set, indicating to the register file that the value in its data output should be written to the register indicated by its register output. The block diagram is available in Figure 7.

REGFILE

This is not a pipeline stage – it is where the pipeline's registers and forwarding hardware live. It contains four registers, which each store a 16-bit value. It takes in a 16-bit value as input, plus a 2-bit register identifier and a `write_enable` flag. When the flag is set, the input value is written to the register specified. The output of each register is connected to a forwarding unit that takes in each stage's `fu_avail` flag, its destination register, and its output value. If the flag is set and the destination register matches, then the output from the forwarding unit is replaced with the output from that pipeline stage. Priority is given to stages earlier in the pipeline, since they hold the values returned by the most recent instruction to enter the pipeline. The net result is that the registers appear to have the values set before the WB stage. The block diagram of the register file is provided in Figure 8.

Instruction Set

The instruction set I implemented is very basic – it contains three branching instructions, nine arithmetic instructions, and three load/store instructions. Each instruction is 22 bits long, starting with a 4-bit opcode. For conditional branching instructions, the next two bits contain the register number used for the branching condition, and the next 16 bits contain the address of the instruction to jump to. The unconditional branch omits the register and goes directly from the opcode to the instruction. The first two bits of a store instruction, after the opcode, contain the number of the register where the value to be stored is located, and the next two bits contain the register number of the value to be stored. A load instruction is the same, but the first register is the register to store the value in after it is loaded. The load immediate instruction contains a register identifier followed by a 16-bit immediate value to store in the register. For arithmetic instructions, the opcode is followed by three registers – the first is where the result should be stored, and the second and third are the operands for the instruction. The full table of instructions is provided in Figure 9.

Assembler

My assembler was written in python. It takes three passes plus a "zeroth" pass to compile a binary for your program. The "zeroth" pass just splits each instruction into tokens so that it can be processed more easily in later passes. I refer to this as a "zeroth" pass because it could easily be integrated into the next step, but I left it on its own to make the code easier to understand. The "zeroth" pass also identifies the aliases, initial memory values, and loop addresses indicated in the code and stores them into a lookup table.

The first pass identifies dependencies between instructions. Whenever an instruction reads from a register, it marks the last instruction to write to it as a dependency. Whenever an instruction writes to a register, it marks the last instructions to both read and write to it as an anti-dependency. These will be important when reordering instructions to minimize stalls.

The second pass identifies places where a NOOP instruction needs to be inserted. Every branch instruction must have two stalls inserted to prevent any code from executing until the branch passes the EX stage, moving the program counter to the appropriate location. Load instructions must have one NOOP inserted to ensure the register is properly updated before it is read. The Load stall can be omitted if the next instruction is not dependent on the load – essentially it acts as a stall itself. In cases where the stall is necessary, the assembler will search for a valid instruction to move after the load to prevent the stall. There are a few criteria that it looks for when choosing a candidate:

1. If the instruction came from before the load, all the instructions between it and the load instruction must be neither dependent nor anti-dependent on it
2. If the instruction came from after the load, it must not be dependent or anti-dependent on any of the instructions between it and the load
3. All load and store instructions must remain in the same relative order
4. No instruction may cross a boundary set by a loop instruction or a loop address

When an appropriate instruction is found, it is moved after the loop and the stall has been avoided. If no instruction is found, then a NOOP is inserted. When a NOOP is inserted either for a branching instruction or for a load instruction, all branch addresses after the NOOP are incremented to account for the extra instruction.

Code for the assembler has three sections that can appear in any order. The first is the alias section, where aliases can be defined to represent literal hex values. An alias definition looks like this:

```
$ALIAS 0x1234
```

The next section is the data section, where data memory can be initialized. A memory initialization includes a hex address followed by a hex value, either of which can be an alias. The file is parsed from top to bottom, so to use an alias, the alias must be defined above its use location. Memory initialization looks like this:

```
$ALIAS 0x1234
```

The final section is the start section, where code resides. Each line has an an instruction followed by its operands. Registers are given as a percent sign followed by the register number, from 0 to 3. Literal values can be passed in hex or can be identified by an alias. Branch instructions must jump to a named branch location, identified by placing a name beginning with a dot above the target instruction. Some instructions might look like this:

```
.LOOP
ADD     %0 %0 %1
LOADI   %1 $ALIASVAL
LOADI   %2 0xABCD
STORE   %0 %2
JMP     .LOOP
```

Whitespace is ignored, so both tabs and spaces are acceptable. Each section starts by being identified by name like this:

```
#alias
#data
#start
```

The assembler outputs data memory and instruction memory initialization files and has a `-c` command line flag to print out a list of ModelSim commands to program the memory directly.

Testing Methodology

While writing the VHDL, I tested each pipeline stage individually. I used the wave viewer in ModelSim to set the inputs to values I could predict the output of, and stepped through the execution until I was either satisfied that it was working as expected, or I noticed a problem area and could troubleshoot it more effectively. Once I had each stage working as I wanted, I created one top-level assembly and connected the stages together. I wrote some instructions by hand, and inserted them into instruction memory to make sure they executed the way I expected them to, and if they didn't I looked closer at the stage where they failed and the connections to and from that stage.

When the VHDL was done, I tested my assembler by writing simple programs to ensure the instructions came out correctly. When they were in a working state, I wrote some programs that I knew could be rearranged to avoid stalls tested that the assembler rearranged them properly. I set the assembler to print out the human-readable versions of the instructions after reordering to see the results more easily. When that was complete, I wrote a few more complex programs and loaded them into the pipeline to ensure their output was as expected.

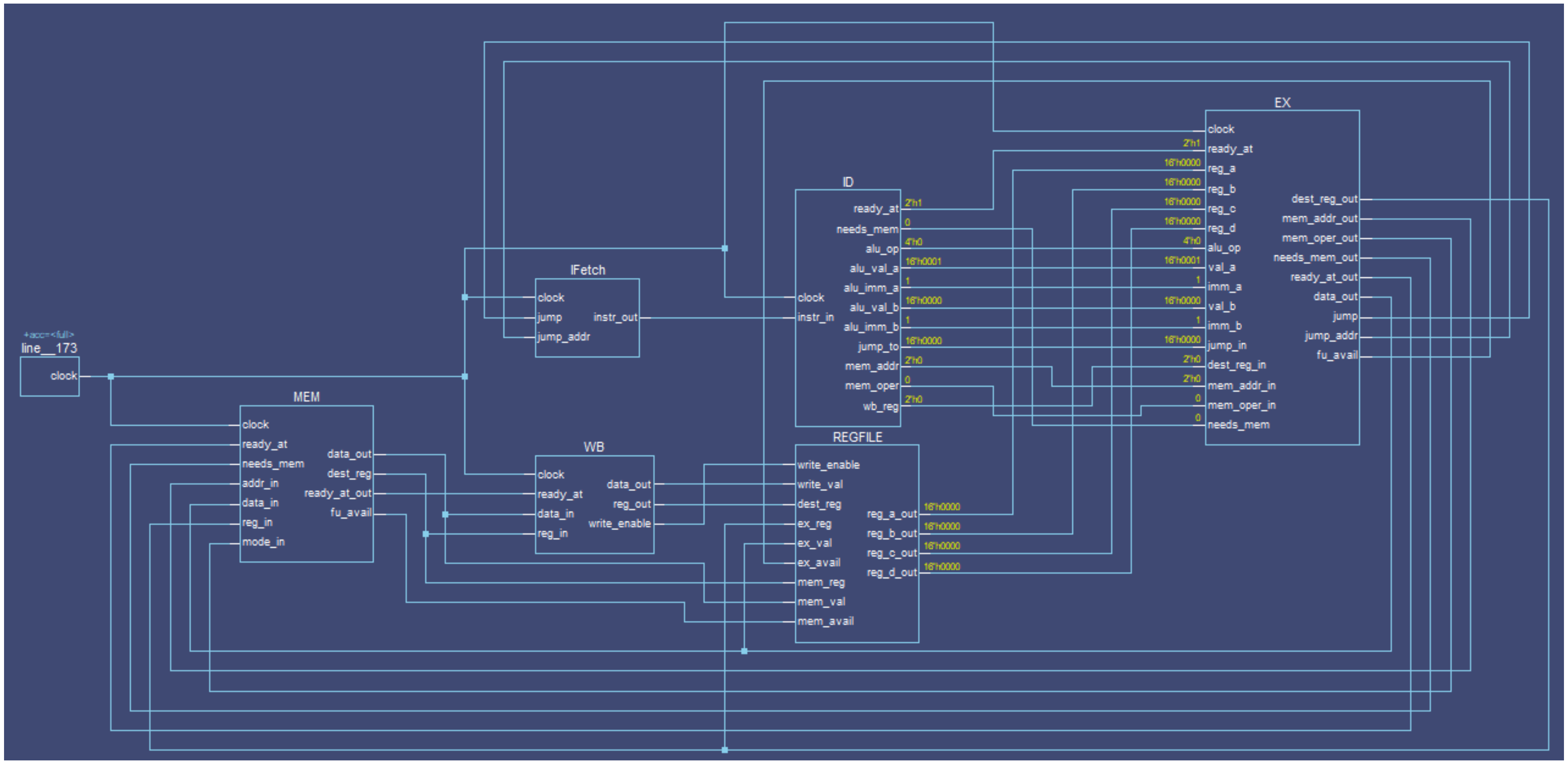
Takeaways

I felt that this project was a fun way to get hands-on experience with the RISC pipeline. Before I started I understood how it worked conceptually, but I ran into a lot of roadblocks on the way to my final implementation that exposed plenty of tiny details that I had to learn how to deal with. The specific details of how registers are forwarded from intermediate stages in the pipeline was something that took me two or three iterations to nail down.

Learning how to use VHDL was perhaps the biggest hurdle in this project. The first week or so was spent just trying to get a good understanding of how to use it effectively, and I think I still have a way to go before I am using it efficiently. I could see myself continuing to use it in the future for hobby projects at home.

The implementation of the assembler was easier than I anticipated, but I did have some issue in determining how to reorder instructions to prevent stalls after load instructions. Ultimately, I came up with the four criteria in the Assembler section that let me safely reorder instructions. The first two were straightforward to figure out from the definitions of dependency and anti-dependency. The third criterion was also simple, because if they get reordered then memory consistency is violated. I spent some time thinking I could rearrange them if the destination address is different, but I soon realized that I cannot know the destination address without running the program. The fourth criterion exists because moving an instruction across any branch boundaries means it is being added to or removed from a loop or is being moved to a different execution path and thus changes the behavior of the program. There are some instances where it might be safe to do so but detecting them would require more advanced code introspection.

My biggest regret in the project was designing the pipeline with only four general-purpose registers. This strongly limits the ability to write efficient programs. Combined with the lack of immediate values in arithmetic instructions, this limitation leads to excessive use of main memory. In my proposal, I hoped that fewer registers would simplify the design, but after completing the project I see that I could easily drop in as many as I wanted. I had hoped to keep the instruction size to 16 bits, but since it already had to expand to 22 bits I may as well have added one extra bit and used eight registers. I might extend the code in the future to do so.


line__173
clock
MEM
clock
ready_at
needs_mem
addr_in
data_in
reg_in
mode_in
data_out
dest_reg
ready_at_out
fu_avail
IFetch
clock
jump
jump_addr
instr_out
WB
clock
ready_at
data_in
reg_in
data_out
reg_out
write_enable
ID
clock
instr_in
ready_at
needs_mem
alu_op
alu_val_a
alu_imm_a
alu_val_b
alu_imm_b
jump_to
mem_addr
mem_oper
wb_reg
REGFILE
write_enable
write_val
dest_reg
ex_reg
ex_val
ex_avail
mem_reg
mem_val
mem_avail
reg_a_out
reg_b_out
reg_c_out
reg_d_out
EX
clock
ready_at
reg_a
reg_b
reg_c
reg_d
alu_op
val_a
imm_a
val_b
imm_b
jump_in
dest_reg_in
mem_addr_in
mem_oper_in
needs_mem
dest_reg_out
mem_addr_out
mem_oper_out
needs_mem_out
ready_at_out
data_out
jump
jump_addr
fu_avail
Figure 1

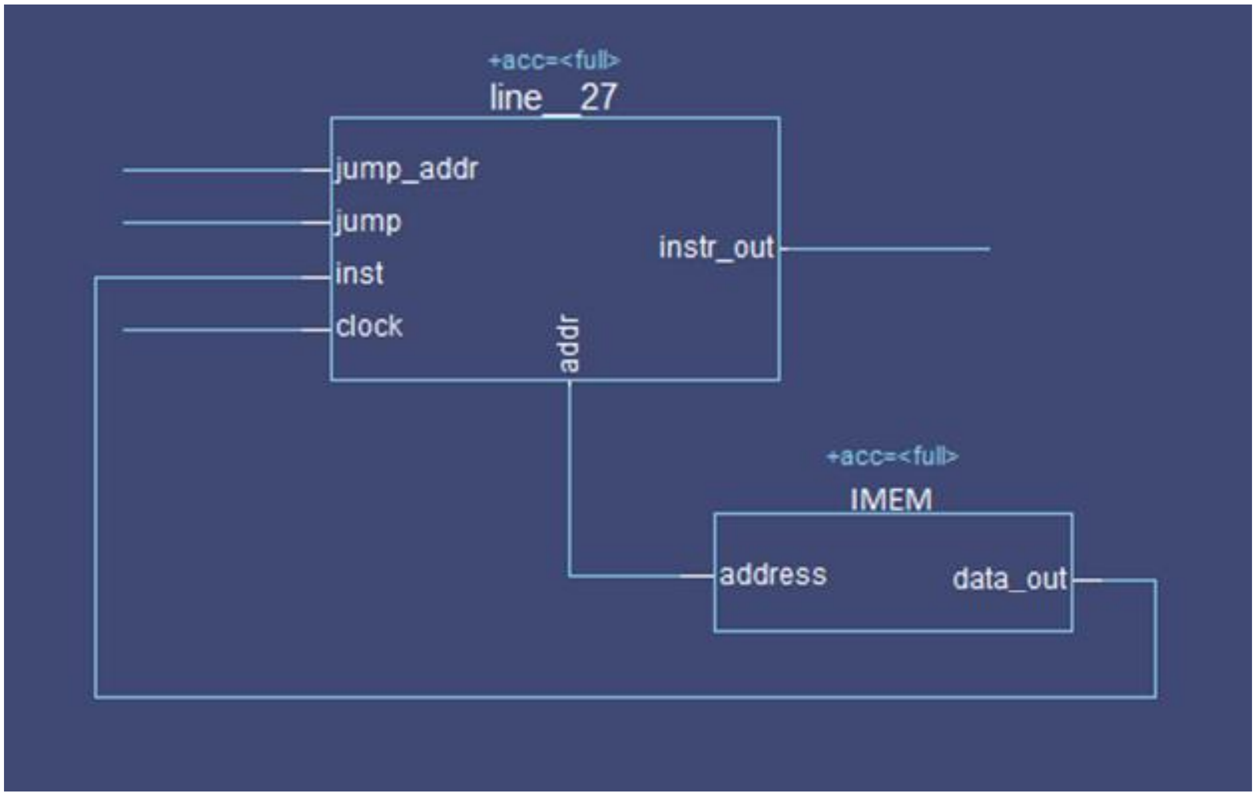


Figure 3

alu_imm_a	A flag indicating whether alu_val_a is a register number (0) or an immediate value (1)
alu_imm_b	Same as above, but for alu_val_b
alu_op	A 4-bit opcode that indicates the ALU operation. Different from the instruction opcode.
alu_val_a	The value for input A of the ALU. May be a register number or an immediate value
alu_val_b	Same as above, but for input B
jump_to	The memory address to jump to if the jump comparison succeeds
mem_addr	The memory address where data should be read from or written to
mem_oper	A flag indicating whether we are reading from (0) or writing (1) to memory
needs_mem	A flag indicating if the instruction does (1) or does not (0) require memory access
ready_at	A 2-bit flag indicating when the results of the instruction will be available for forwarding. After the EX stage (01), After the MEM stage (10), or never (00)
wb_reg	The register number where the final result will be stored.

Figure 2

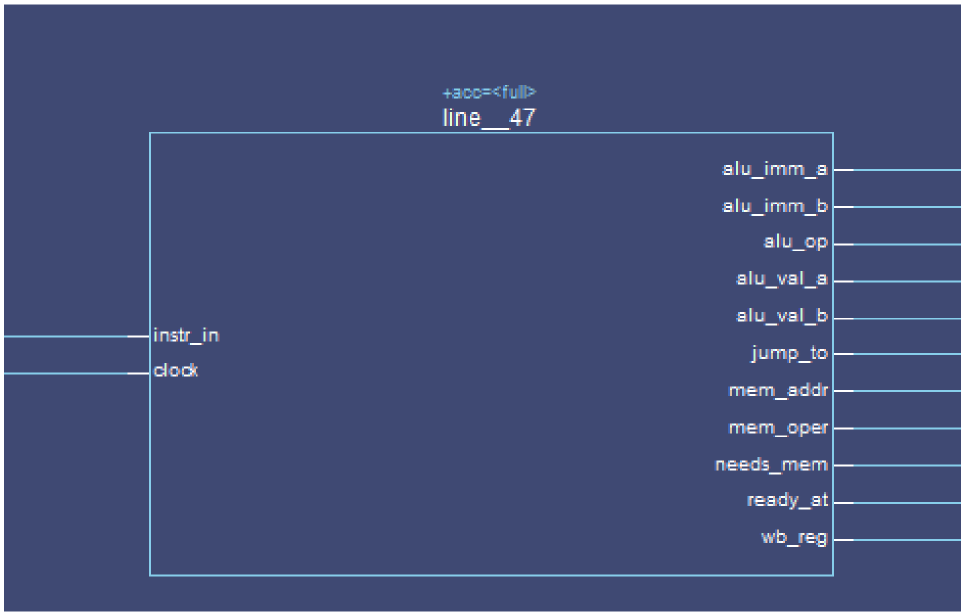


Figure 4

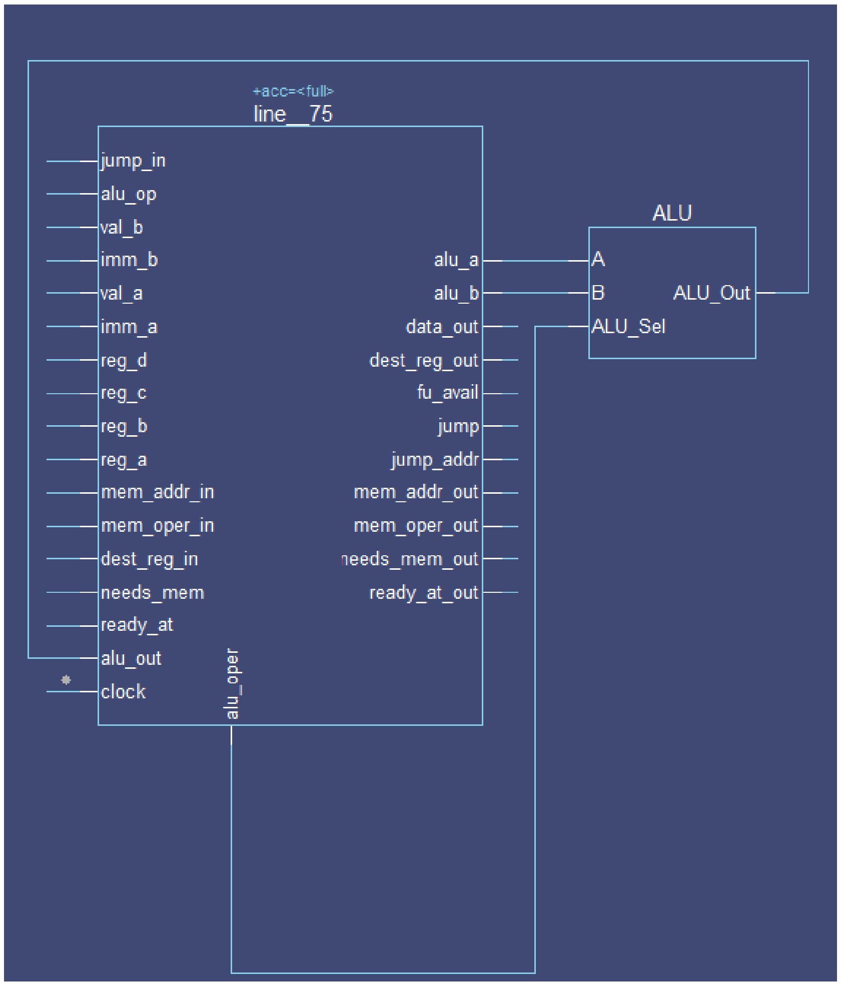


Figure 5

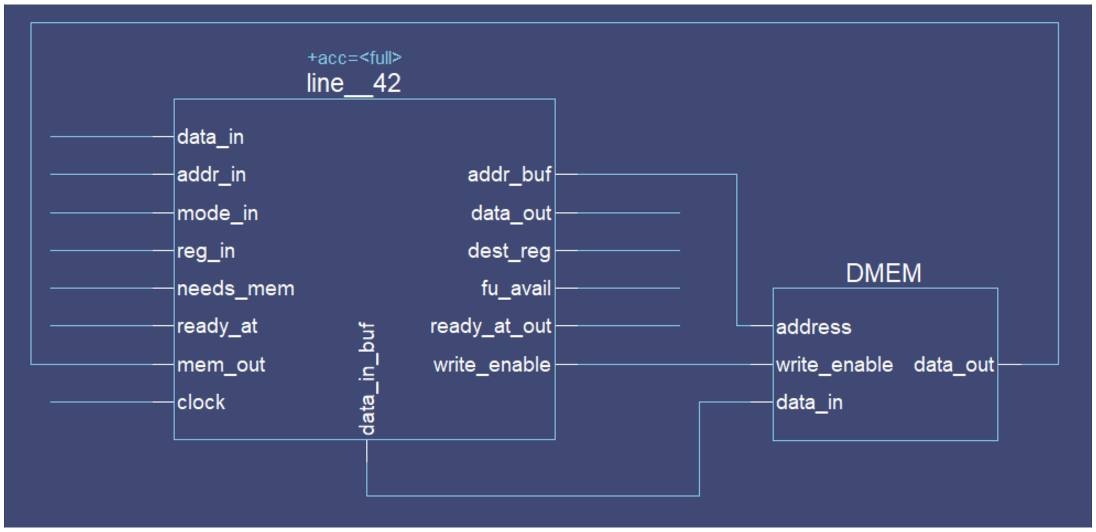


Figure 6

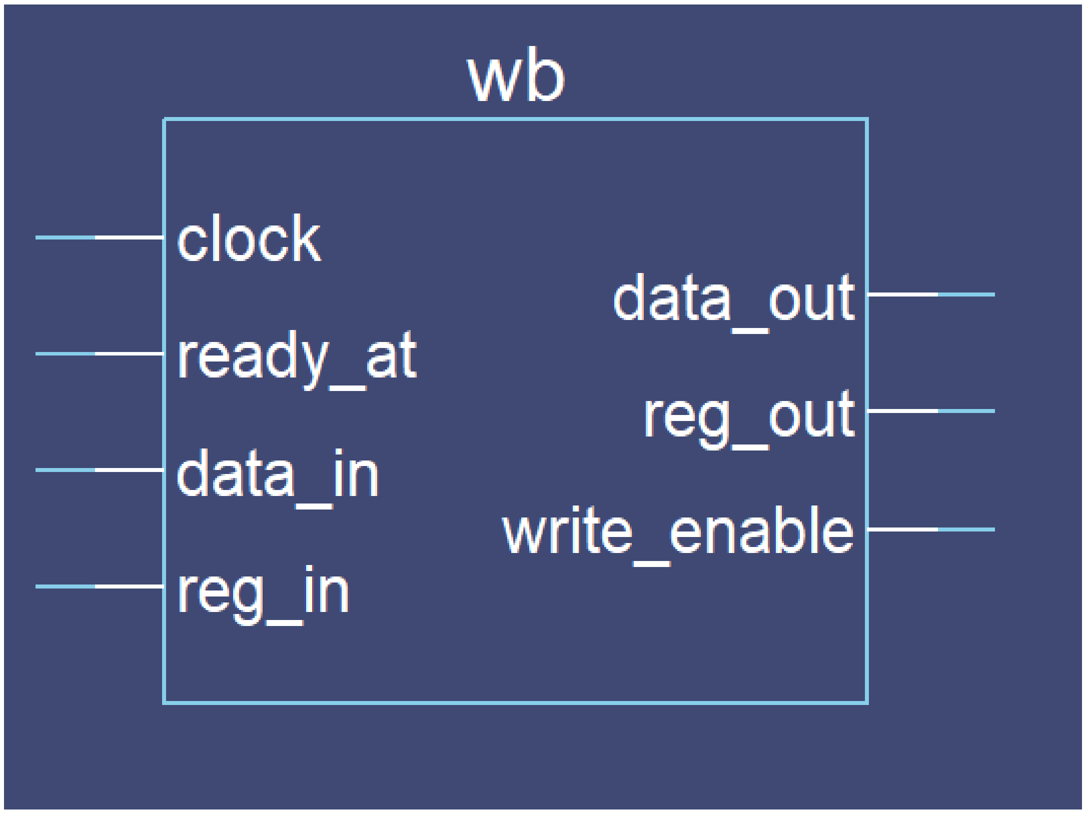


Figure 7

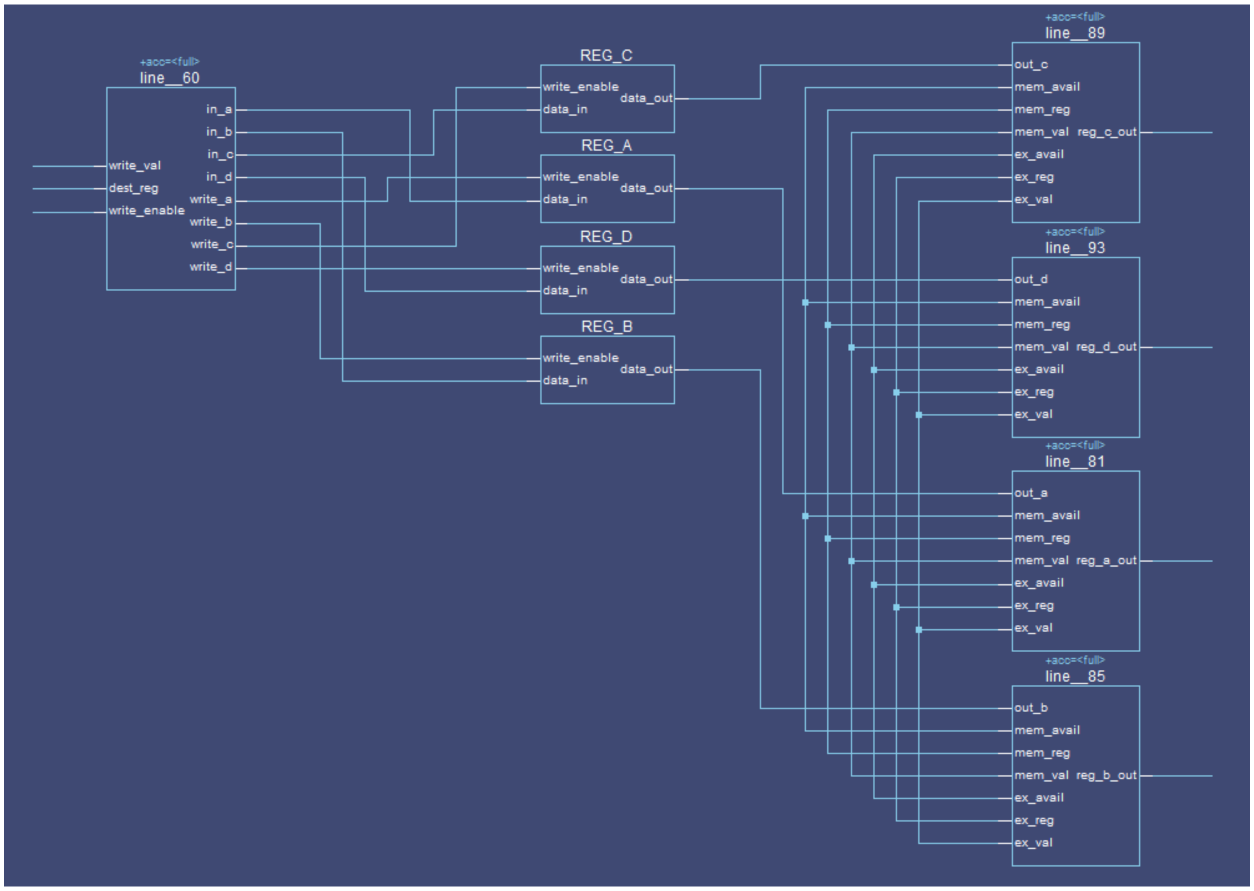


Figure 8

Instruction	OPCODE	Data		
NOOP	0000000000000000000000			
JMP	0001	Instruction number		
STORE	0010	Src	Mem Address	
JNZ	0011	Src	Instruction number	
JEZ	0100	Src	Instruction number	
LOAD	0101	Dest	Mem Address	
LOADI	0110	Dest	Value	
ADD	0111	Dest	Src 1	Src 2
SUB	1000	Dest	Src 1	Src 2
MUL	1001	Dest	Src 1	Src 2
DIV	1010	Dest	Src 1	Src 2
AND	1011	Dest	Src 1	Src 2
OR	1100	Dest	Src 1	Src 2

Figure 9