MADE BY: Ahmed Emad

CPU

Instruction set

Data Manipulation

• Add	x”00”	A + B
• Addf	x”01”	A + B (Float)
• Sub	x”02”	A – B
• Subf	x”03”	A – B (Float)
• Mult	x”04”	A x B
• Multf	x”05”	A x B (Float)
• Div	x”06”	A / B
• Divf	x”07”	A / B (Float)
• And_AB	x”08”	A . B
• Or_AB	x”09”	A + B
• ComE	x”0A”	A = B? 1 or 0
• ComL	x”0B”	A > B? 1 or 0
• ComG	x”0C”	A < B? 1 or 0
• ComE_F	x”0D”	A = B? 1 or 0 (Float)
• ComL_F	x”0E”	A > B? 1 or 0 (Float)
• ComG_F	x”0F”	A < B? 1 or 0 (Float)
• Not_A	x”10”	! A
• Rand_s	x”11”	Random number with seed
• Rand	x”12”	Random number
• Print	x”13”	Print in GPU
• floatToInt	x”14”	Change A float to int
• intToFloat	x”15”	Change A int to float

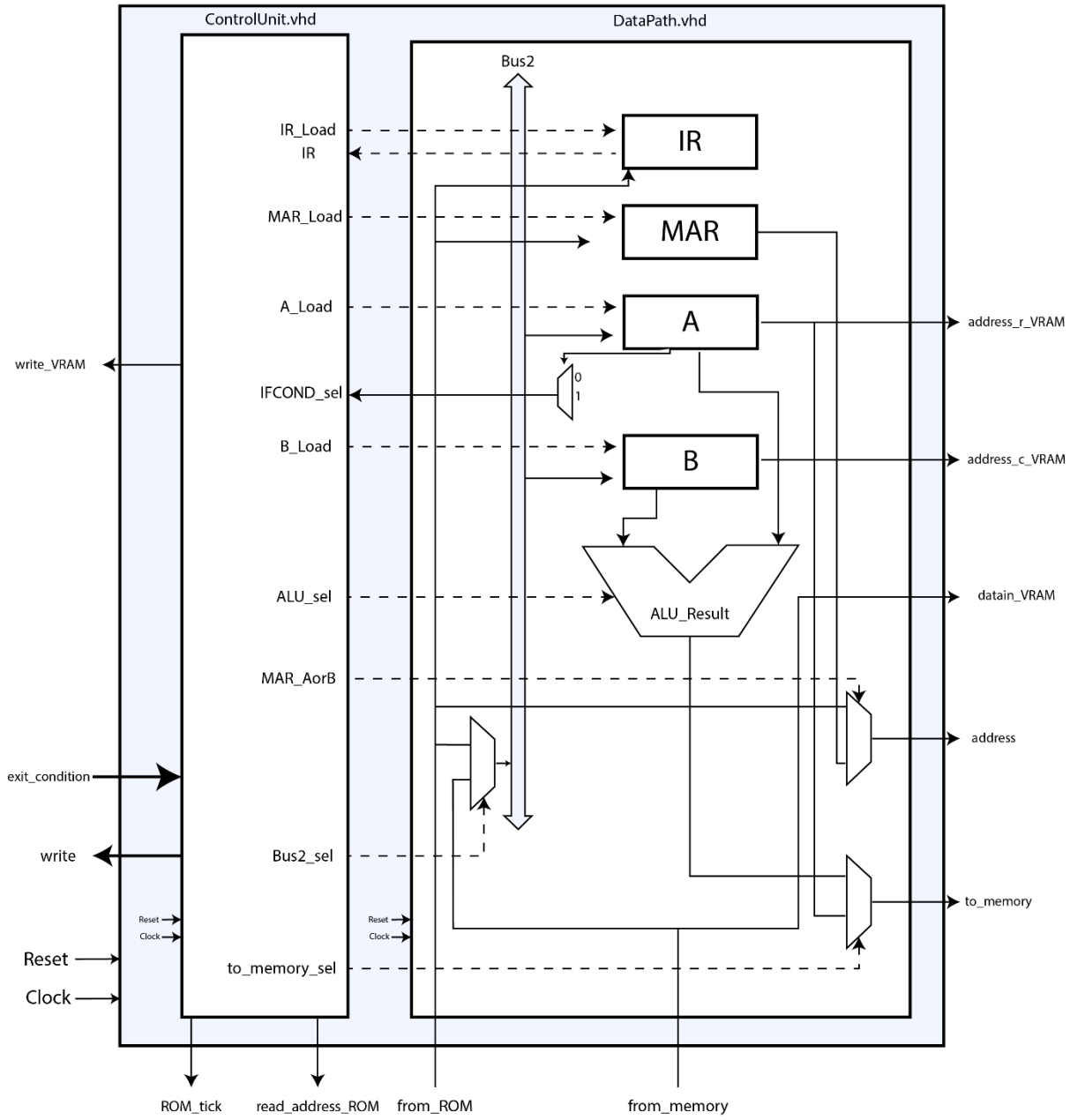
other operations

• if_cond	x”16”	If (o or 1)? Address_true Address_false
• Copy	x”17”	Copy value of address 1 to address 2
• Save	x”18”	Save value to address
• goto	x”19”	Go to certain line of code in ROM
• VRAM_SAVE	x”1A”	Save value to address in VRAM’
• Idle_till_signal	x”1B”	make system Idle while GPU finishes

CPU Implementation

The CPU contains two components, the control unit (control_unit.vhd) and the data path (data_path.vhd). The data path contains all of the registers and the ALU. The ALU is implemented as a sub-component within the data path (alu.vhd). The data path also contains a bus system in order to facilitate data movement between the registers and memory. The bus system is implemented with two multiplexers that are controlled by the control unit. The control unit contains the finite-state machine that generates all control signals for the data path as it performs the fetch-decode execute steps of each instruction.

Block Diagram



Data Path Implementation

The data path consists mainly of registers and ALU, controlled by different signals from control unit which either control registers or MUX sending data to buses or other components. Data enter from_ROM signal and is distributed among IR register, MAR register, bus signal or address signal depending on loads. Then depending on value of IR register computer goes to different states where data take different routes as shows later in detailed execution of every command.

Control Unit Implementation

We'll first look at the formation of the VHDL to model the FSM and then turn to the detailed state transitions in order to accomplish a variety of the most common instructions. The control unit sends signals to the data path in order to move data in and out of registers and into the ALU to perform data manipulations. The finite-state machine is implemented with the behavioral modeling techniques. The model contains three processes in order to implement the state memory, next state logic, and output logic of the FSM. User-defined types are created for each of the states defined in the state diagram of the FSM.

```
type FSM is (S_DECODE_0,
                S_LOADA_1, S_LOADB_2, S_DATA_MAN_2,
                S_COPY_1,
                S_SAVE_1,
                S_GOTO_1,
                S_VRAMSAVE_1, S_VRAMSAVE_2, S_VRAMSAVE_3, S_VRAMSAVE_4, S_VRAMSAVE_5,
                S_IFCOND_1, S_IFCOND_2, S_IFCOND_3,
                S_IDLE _1, S_IDLE _2
                                    );
```

Within the architecture of the control unit model, the state memory is implemented as a separate process that will update the current state with the next state on each rising edge of the clock. The reset state will be the first fetch state in the FSM (i.e., S_DECODE_0). The following VHDL shows how the state memory in the control unit can be modeled:

```
process(clock,reset)
    begin
       if(reset='0') then
          current_state <= S_DECODE_0;
       elsif(rising_edge(clock)) then
          current_state <= next_state;
       end if;
    end process;
```

Finally, the output logic is modeled as a separate process. the process only depends on the current state and IR. The following VHDL shows a portion of the output logic process:

```
process(current_state,IR)
    begin
          IR_Load <= '0';
          MAR_Load <= '0';
          A_Load <= '0';
          B_Load <= '0';
          write <= '0';
          ROM_tick <= '0';
          Bus2_sel <= '0';
          read_address_ROM <='0';
          write_VRAM <= '0';
          VRAM_Load <= '0';

          case(current_state) is
              when S_DECODE_0 =>
                        IR_Load <= '1';
                        MAR_Load <= '1';

                        If (IR >= x"00" and IR <=x"14") then
                                  MAR_AorB <= "01";
                                  A_Load <='1';
                                  next_state <= S_LOADA_1;
                        elsif (IR = if_cond) then
                                  Bus2_sel <= '0';
                                  MAR_AorB <= "01";
                                  A_Load <= '1';
                                  next_state <= S_IFCOND_1;
                        elsif (IR = save) then
                                  Bus2_sel <= '1';
                                  A_Load <= '1';
                                  next_state <= S_SAVE_1;
                        when....
                                  .
                                  .
                                  .
                        when others =>
                                  next_state <= S_DECODE_0;
              end case;
end process;
```

Detailed Execution for commands Example

Add Command