- ## NAND INSTRUCTION:

EXPLANATION:

- Instruction encoding: In this example we are going to employ a reserved opcode / field code:
    1. Specifically, we are going to use the *SPECIAL* opcode / function field *001110*, which is an opcode / function field that is reserved for future use (marked with a "*" in Table A.3 of document "Architecture for Programmers, Volume II-A").
    2. Being reserved, this operation / field code will "cause a *Reserved Instruction Exception*", which we must inhibit:
        a. signal ***ri_e*** is assigned in m14k_mpc_dec, from the OR function of several signals.
        b. Among those signals, one is for *SPECIAL* instruction Reserved Exceptions: signal ***spec_ri_e***. This signal is 1 for all Reserved Function Fields in the SPECIAL Opcode. We just need to comment line ***(mpc_ir_e[5:0] == 6'o16)*** when computing that signal, inhibiting a Reserved Exception for that opcode / field code.

- Inclusion of the functionality for the new instruction in the processor. Figure 1 shows the original Logic Unit, while Figure 2 shows the modified Logic Unit after including support for the new instruction. All changes related to the new instruction are tagged with comment: *// NAND* in the soft-core. We must perform the following actions:
    1. We must include support for the new instruction in the following two signals:
        a. ***sel_logic_e***:
            - This signal controls registration of the output of logic operations from Stage-E to Stage-M.
            - This signal is assigned in module m14k_mpc_dec. We just include the opcode / field code of the new instruction: *(mpc_ir_e[5:0] == 6'b001_110)*.
        b. ***alu_sel_e***:
            - This signal must be 1 for Arithmetic-Logic operations, for selecting, at Stage-M, the output of the ALU.
            - This signal is assigned in module mpc_dec. Like in the previous signal, we just include the opcode / field code of the new instruction: *special_e && (mpc_ir_e[5:0] == 6'o16)*.
        c. What about the RF write at W-Stage? For Special Instructions (note that NOP is also an special instruction), only for some of them the RF write is not enabled (line 1250 in module "m14k_mpc_dec"). For the remaining Special instructions, it´s always enabled. So for the new instruction we just do nothing.

    2. We must include a new Functional Unit for calculating the Logic NAND. This is simply achieved by including the following line in the m14k_edp module:
        assign a_nand_b_e [31:0] = ~a_and_b_e; // NAND

3. We must include a new signal (***nand_instruction*** is 1 if we have the new instruction and 0 if we have another instruction) for selecting the nand operation at the output of the Logic Unit.
    a. ***nand_instruction***:
    assign nand_instruction = special_e & (mpc_ir_e[5:0] == 6'o16); // NAND

    b. The following MUX has been modified:
    mvp_mux4 #(32) _logic_out_e_partial_31_0_(
    logic_out_e_partial[31:0], // Output
    mpc_alufunc_e, // Control Signal
    a_and_b_e, // Input-1
    a_or_b_e, // Input-2
    a_xor_b_e, // Input-3
    a_nor_b_e); // Input-4

    c. The following MUX has been included:
    mvp_mux2 #(32) _logic_out_e_31_0_(
    logic_out_e[31:0], // Output
    nand_instruction, // Control Signal
    logic_out_e_partial, // Input-1
    a_nand_b_e); // Input-2

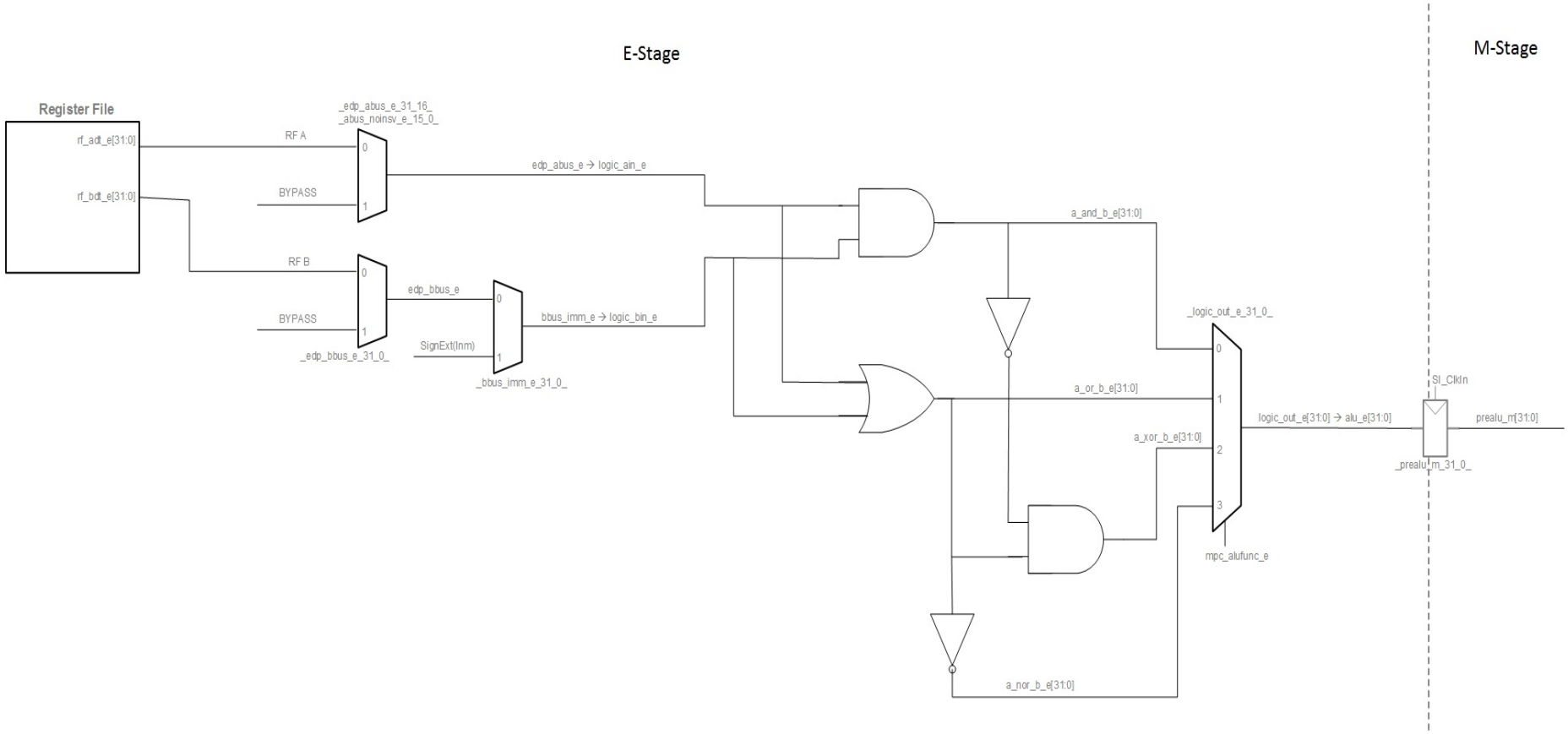

E-Stage
M-Stage
Register File
rf_adt_e[31:0]
rf_bdt_e[31:0]
_edp_abus_e_31_16_
_abus_noinsv_e_15_0_
RF A
BYPASS
edp_abus_e → logic_ain_e
RF B
BYPASS
edp_bbus_e
_edp_bbus_e_31_0_
SignExt(Inm)
_bbus_imm_e_31_0_
bbus_imm_e → logic_bin_e
a_and_b_e[31:0]
a_or_b_e[31:0]
a_xor_b_e[31:0]
a_nor_b_e[31:0]
_logic_out_e_31_0_
mpc_alufunc_e
logic_out_e[31:0] → alu_e[31:0]
SI_ClkIn
prealu_m[31:0]
_prealu_m_31_0_

FIGURE 1. Original Logic Unit.

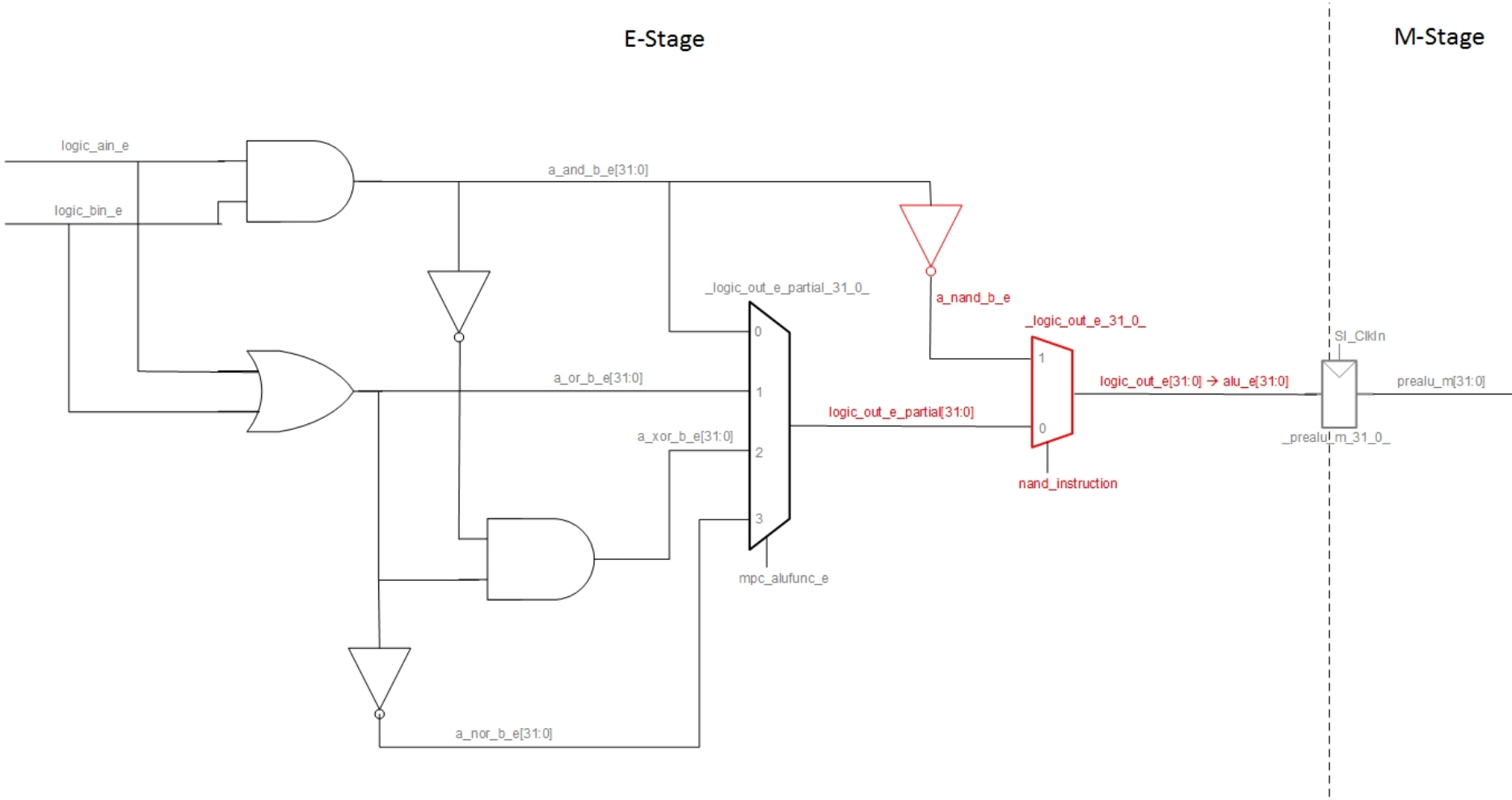
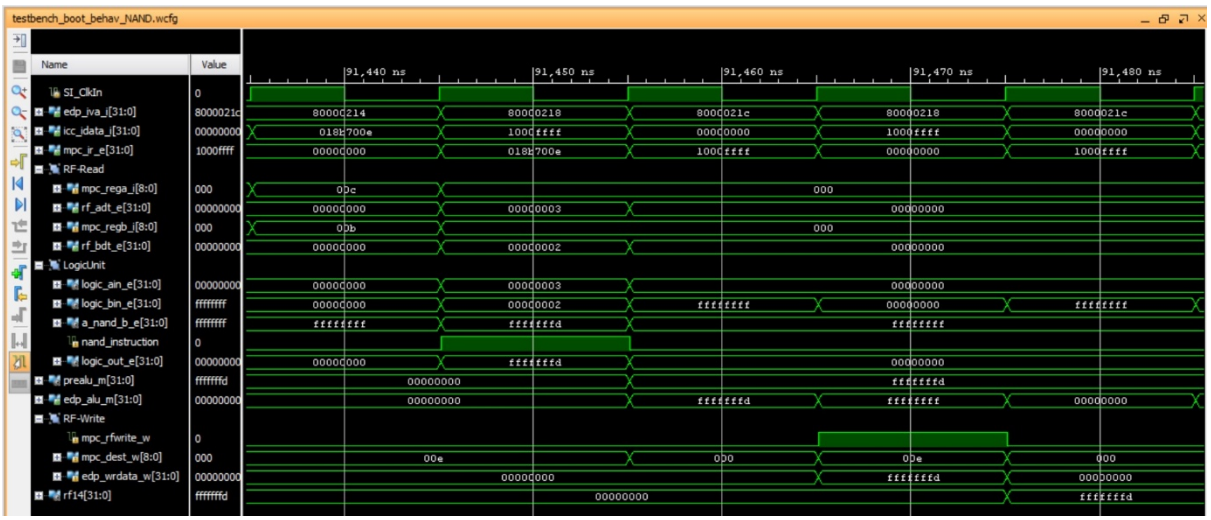


FIGURE 2. Modified Logic Unit. The new Hw and the new Signals are highlighted in red. Note that we have renamed some signals or muxes (also highlighted).

EXAMPLE - SIMULATION:



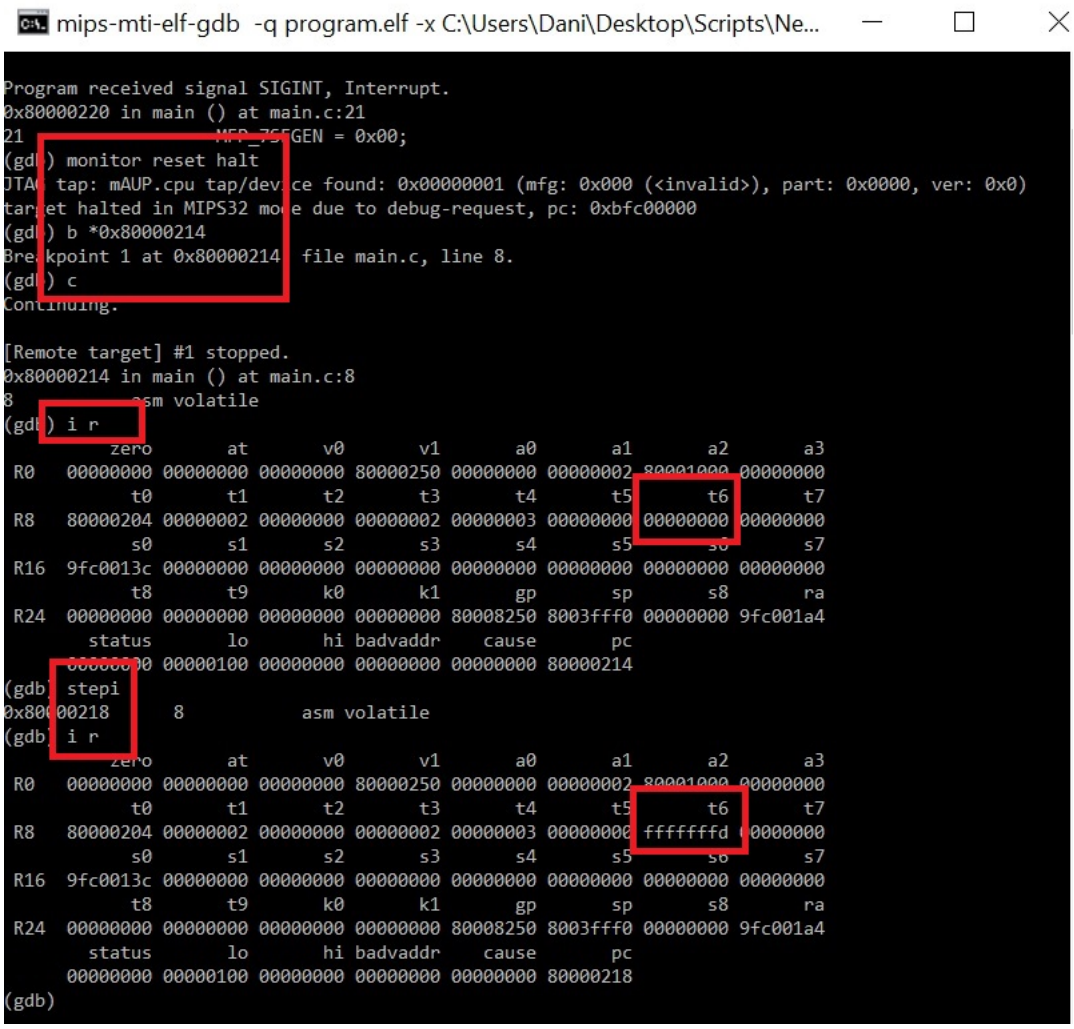
Observe that, in the 5th cycle, the destination register (rf14=$t6) is written with the result of the nand operation (0xfffffffd).

EXAMPLE - IMPLEMENTATION IN THE FPGA:

When the program is downloaded on the board, you should see on the 7-seg displays:

- 7-seg displays=$t6, which in our example is 0xfffffffd

Then, when you debug the program following the steps stated in the document, you should observe the following:


mips-mti-elf-gdb -q program.elf -x C:\Users\Dani\Desktop\Scripts\Ne...
Program received signal SIGINT, Interrupt.
0x80000220 in main () at main.c:21
21 WDT_TESTGEN = 0x00;
(gdb) monitor reset halt
JTAG tap: mAUP.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>), part: 0x0000, ver: 0x0)
target halted in MIPS32 mode due to debug-request, pc: 0xbfc00000
(gdb) b *0x80000214
Breakpoint 1 at 0x80000214: file main.c, line 8.
(gdb) c
Continuing.
[Remote target] #1 stopped.
0x80000214 in main () at main.c:8
8 asm volatile
(gdb) i r
zero at v0 v1 a0 a1 a2 a3
R0 00000000 00000000 00000000 80000250 00000000 00000002 80001000 00000000
t0 t1 t2 t3 t4 t5 t6 t7
R8 80000204 00000002 00000000 00000002 00000003 00000000 00000000 00000000
s0 s1 s2 s3 s4 s5 s6 s7
R16 9fc0013c 00000000 00000000 00000000 00000000 00000000 00000000 00000000
t8 t9 k0 k1 gp sp s8 ra
R24 00000000 00000000 00000000 00000000 80008250 8003fff0 00000000 9fc001a4
status lo hi badvaddr cause pc
00000000 00000100 00000000 00000000 00000000 80000214
(gdb) stepi
0x80000218 8 asm volatile
(gdb) i r
zero at v0 v1 a0 a1 a2 a3
R0 00000000 00000000 00000000 80000250 00000000 00000002 80001000 00000000
t0 t1 t2 t3 t4 t5 t6 t7
R8 80000204 00000002 00000000 00000002 00000003 00000000 fffffffd 00000000
s0 s1 s2 s3 s4 s5 s6 s7
R16 9fc0013c 00000000 00000000 00000000 00000000 00000000 00000000 00000000
t8 t9 k0 k1 gp sp s8 ra
R24 00000000 00000000 00000000 00000000 80008250 8003fff0 00000000 9fc001a4
status lo hi badvaddr cause pc
00000000 00000100 00000000 00000000 00000000 80000218
(gdb)