EXPLANATION:

- Let´s first understand the original instructions, MOVZ/MOVN (Figure 3): These instructions are handled like an AND instruction, with some changes:
  - The following signals determine that a MOVZ/MOVN instruction is being executed (module m14k_mpc_dec):
    1. ***mpc_cmov_e***:
       a. 1 if MOVZ or MOVN instruction.
       b. 0 if any other instruction.
    2. ***cmov_type_e***:
       a. 1 if MOVZ.
       b. 0 if MOVN.
    3. ***sel_logic_e***: Select logic output.
       a. 1 if MOVZ/MOVN (plus other instructions).

  - Comparison with 0 (module m14k_edp):
    1. ***edp_cndeq_e***: Comparator (used also for other instructions):

       *assign **edp_cndeq_e** = **acmp_e** == **edp_bbus_e**;*

       - ***edp_bbus_e*** is ***Rt***, which is provided from the RF or from a Bypass in case of dependency.
       - ***acmp_e***, for MOVZ/MOVN instructions (i.e. **mpc_cmov_e**=1), is set to *32'h0*:

         mvp_mux2 #(32)
         _acmp_e_31_0_(acmp_e[31:0],
         mpc_cmov_e,
         edp_abus_e,
         32'h0);

  - If the condition is met (***edp_cndeq_e*** == 1):
    The functionality of an AND instruction is used (see example illustrating an AND instruction), with an important change: ***logic_ain_e*** (Source-A) = ***logic_bin_e*** (Source-B) = ***Rs***. Thus, the result of ***logic_ain_e*** & ***logic_bin_e*** = ***Rs***.
    a. Source-A is assigned as in the AND instruction.
    b. Source-B is assigned through a MUX that uses as control signal ***mpc_alubsrc_e***, which depends on ***mpc_cmov_e***:
       - 1st input (selected when ***mpc_cmov_e***==1): ***edp_abus_e***
       - 2nd input (selected when logic instruction, etc.): ***bbus_imm_e***

  - If the condition is not met (***edp_cndeq_e*** != 1):
    kill_cmov_e: This signal is set to 1 if the condition for MOVZ/MOVN is not met:

    assign kill_cmov_e = mpc_cmov_e & (edp_cndeq_e ^ cmov_type_e) **|** *…*

    When ***kill_cmov_e*** is 1, the RF write is inhibited:

    ! kill_cmov_e** → **mvd_e** → **pvd_m** → **pvd_w** → **mpc_rfwrite_w
    (Register file write enable)

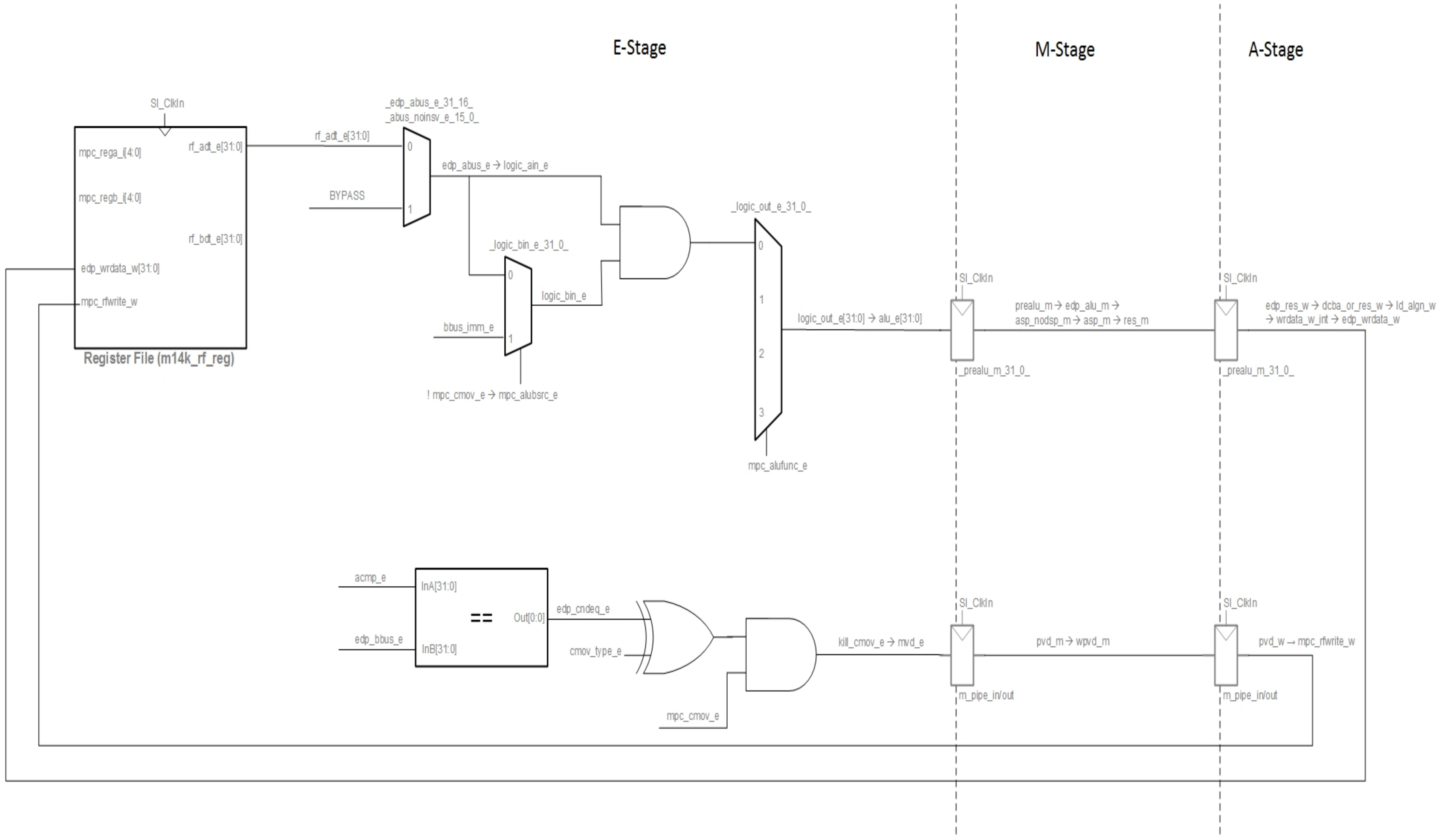


FIGURE 3. Involved structures and signals for MOVZ/MOVN.

- Changes to the original instructions, MOVZ/MOVN, for transforming them into SELEQZ/SELNEZ (Figure 4). All changes related to the new instruction are tagged with comment: *// SELEQZ/SELNEZ* in the soft-core:
  - Modify ***kill_cmov_e*** computation:
    1. In the original instructions, when the condition is not met, ***kill_cmov_e*** is set, which inhibits RF write in Stage-W.

       kill_cmov_e = *mpc_cmov_e & (edp_cndeq_e ^ cmov_type_e)* **|** *mpc_movci_e & ~cp1_btaken;*
    2. In the new instructions, RF must be written in all cases.

       kill_cmov_e = *mpc_movci_e & ~cp1_btaken;*
  - Create new signal for distinguishing between condition met / not met:

    cond_not_met_e = *mpc_cmov_e & (edp_cndeq_e ^ cmov_type_e)*

    This signal is computed at m14k_mpc_ctl and passed to m14k_edp.
  - A new MUX is included in m14k_edp for selecting 0s as the Source-B for the AND operation, when ***cond_not_met_e*** is 1. Note that ***cond_not_met_e*** can only be 1 for MOVZ/MOVN instructions.

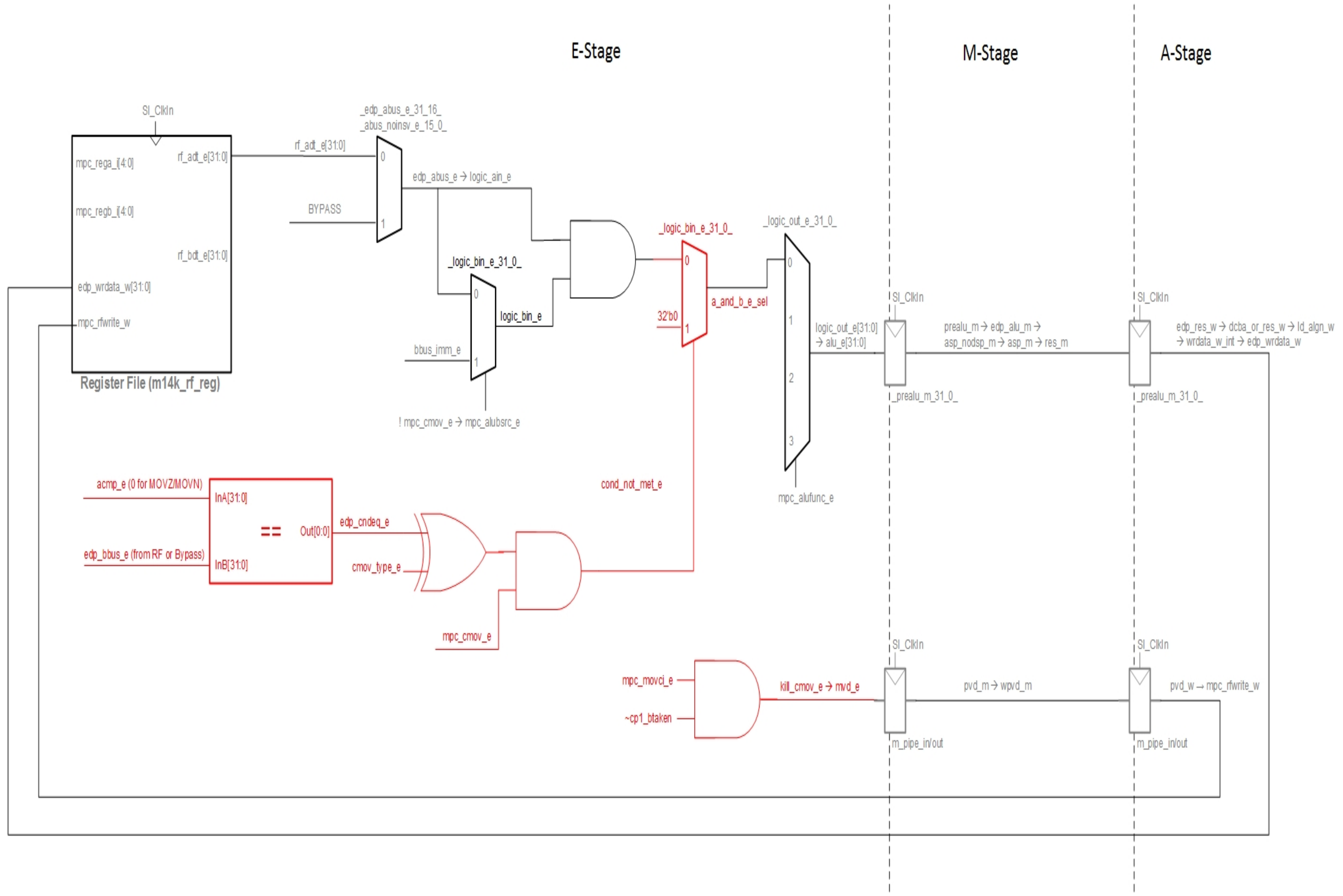
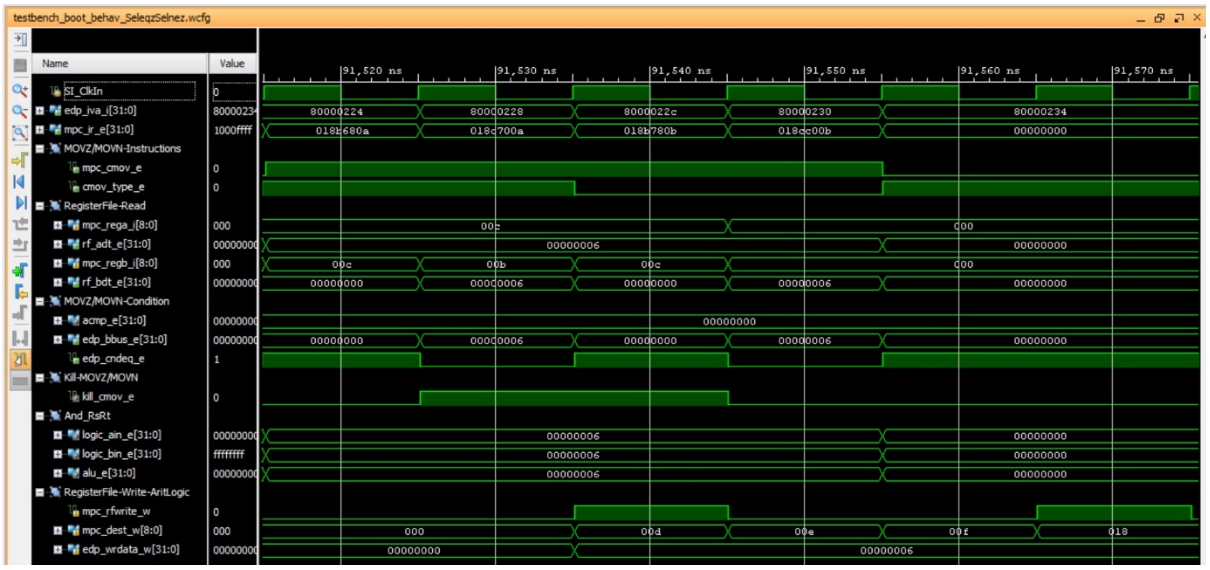


FIGURE 4. Involved structures and signals for SELEQZ/SELNEZ.

<u>EXAMPLE - SIMULATION</u>:

<u>ORIGINAL PROCESSOR</u>:

```
        "   li $t4, 6;"
        "   li $t3, 0;"
        ...
        "   movz $t5,$t4,$t3;"
        "   movz $t6,$t4,$t4;"
        "   movn $t7,$t4,$t3;"
        "   movn $t8,$t4,$t4;"
```



Note that in the third cycle, register 0x00d is written to 0x6, and in the sixth cycle, register 0x018 is written to 0x6. Instead, in cycles fourth and fifth the register file is not written.

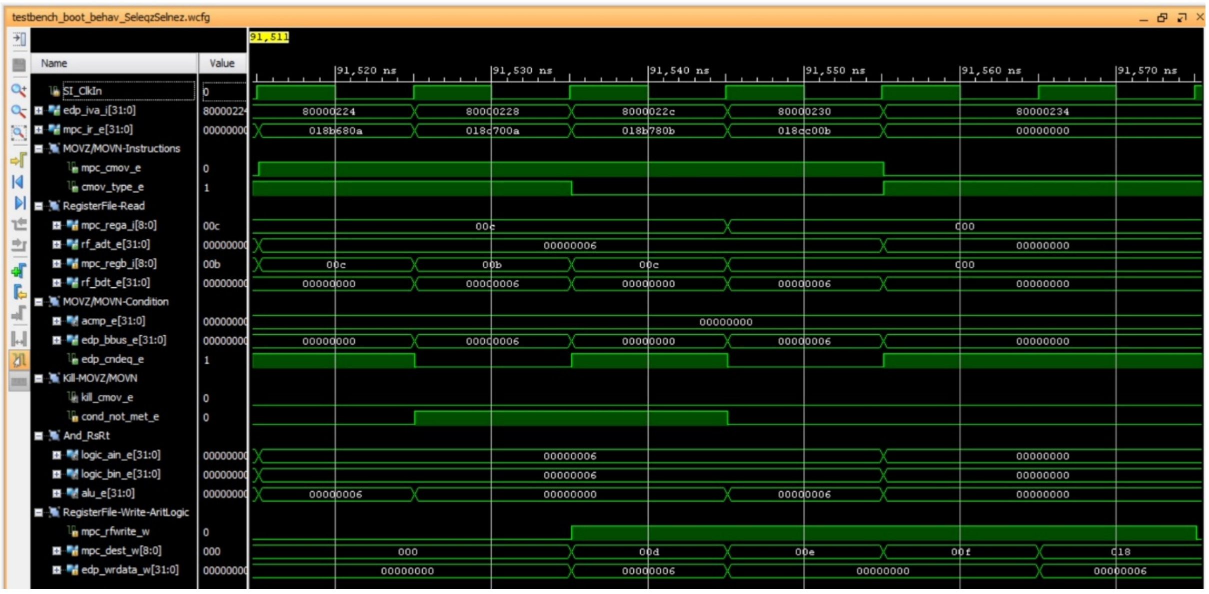
When the program is downloaded on the board, you should see on the 7-seg displays:

- Switches=0 → 7-seg displays=$t5, which in our example is 0x6
- Switches=1 → 7-seg displays=$t6 which in our example is 0x1
- Switches=2 → 7-seg displays=$t7, which in our example is 0x1
- Switches=3 → 7-seg displays=$t8 which in our example is 0x6
- Any other value for the switches → 7-seg displays=0x0

<u>MODIFIED PROCESSOR</u>:

```
        "   li $t4, 6;"
        "   li $t3, 0;"
        ...
        "   movz $t5,$t4,$t3;"
        "   movz $t6,$t4,$t4;"
        "   movn $t7,$t4,$t3;"
```

```
"    movn $t8,$t4,$t4;"
```

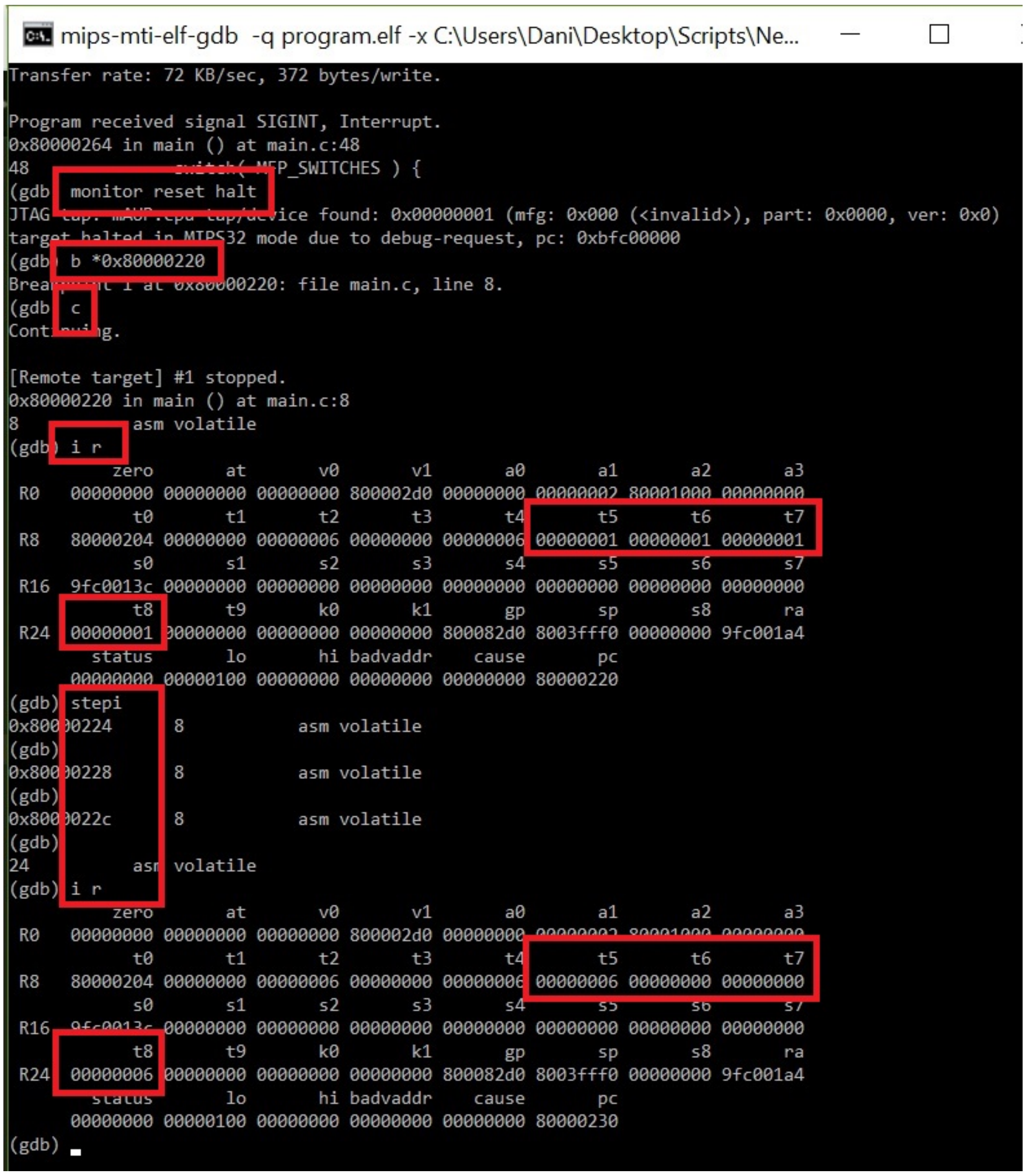


Note that in the third cycle, register 0x00d is written to 0x6, in the fourth cycle, register 0x00e is written to 0x0, in the fifth cycle, register 0x00f is written to 0x0, and in the sixth cycle, register 0x018 is written to 0x6.

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

- Switches=0 → 7-seg displays=$t5, which in our example is 0x6
- Switches=1 → 7-seg displays=$t6 which in our example is 0x0
- Switches=2 → 7-seg displays=$t7, which in our example is 0x0
- Switches=3 → 7-seg displays=$t8 which in our example is 0x6
- Any other value for the switches → 7-seg displays=0x0

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...

```
Transfer rate: 72 KB/sec, 372 bytes/write.

Program received signal SIGINT, Interrupt.
0x80000264 in main () at main.c:48
48                switch( MEP_SWITCHES ) {
(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 *0x80000220
Breakpoint 1 at 0x80000220: file main.c, line 8.
(gdb) c
Continuing.

[Remote target] #1 stopped.
0x80000220 in main () at main.c:8
8           asm volatile
(gdb) i r
          zero       at       v0       v1       a0       a1       a2       a3
 R0   00000000 00000000 00000000 800002d0 00000000 00000002 80001000 00000000
            t0       t1       t2       t3       t4       t5       t6       t7
 R8   80000204 00000000 00000006 00000000 00000006 00000001 00000001 00000001
            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  00000001 00000000 00000000 00000000 800082d0 8003fff0 00000000 9fc001a4
        status       lo       hi badvaddr    cause       pc
      00000000 00000100 00000000 00000000 00000000 80000220
(gdb) stepi
0x80000224      8           asm volatile
(gdb)
0x80000228      8           asm volatile
(gdb)
0x8000022c      8           asm volatile
(gdb)
24          asm volatile
(gdb) i r
          zero       at       v0       v1       a0       a1       a2       a3
 R0   00000000 00000000 00000000 800002d0 00000000 00000002 80001000 00000000
            t0       t1       t2       t3       t4       t5       t6       t7
 R8   80000204 00000000 00000006 00000000 00000006 00000006 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  00000006 00000000 00000000 00000000 800082d0 8003fff0 00000000 9fc001a4
        status       lo       hi badvaddr    cause       pc
      00000000 00000100 00000000 00000000 00000000 80000230
(gdb)
```

<u>EXTRA FUNCTIONALITY:</u>

Change the functionality of the new instructions, so that instead of writing a 0 when the condition is not met, ~Rs is written. I.e:

```
SELEQZ:
    if GPR[rt] = 0      then GPR[rd] ← GPR[rs]
    else                then GPR[rd] ← ~GPR[rs]
SELNEZ:
    if GPR[rt] ≠0   then GPR[rd] ← GPR[rs]
    else                then GPR[rd] ← ~GPR[rs]
```

<u>ANSWER:</u>

The only change is in the new Mux, where, instead of 0s:

```
mvp_mux2 #(32) _and_output_31_0_(
a_and_b_e_sel[31:0],
cond_not_met_e,
a_and_b_e,
32'b0); // SELEQZ/SELNEZ
```

we introduce: ~Rs:

```
mvp_mux2 #(32) _and_output_31_0_(
a_and_b_e_sel[31:0],
cond_not_met_e,
a_and_b_e,
~logic_ain_e); // SELEQZ/SELNEZ
```