Assembly Code Reference

Dot Product

```
.include "nios_macros.s"
.equ     AVECTOR, 0xe00
.equ     BVECTOR, 0xf00
.equ     N, 0xdf0
.equ     DOT_PRODUCT, 0xdf4
.global  _start
_start:
         movia  r2, AVECTOR              /* Register r2 is a pointer to vector A */
         movia  r3, BVECTOR              /* Register r3 is a pointer to vector B */
         movia  r4, N
         ldw    r4, 0(r4)                /* Register r4 is used as the counter for loop iterations */
         add    r5, r0, r0               /* Register r5 is used to accumulate the product */
LOOP:    ldw    r6, 0(r2)                /* Load the next element of vector A */
         ldw    r7, 0(r3)                /* Load the next element of vector B */
         mul    r8, r6, r7               /* Compute the product of next pair of elements */
         add    r5, r5, r8               /* Add to the sum */
         addi   r2, r2, 4                /* Increment the pointer to vector A */
         addi   r3, r3, 4                /* Increment the pointer to vector B */
         subi   r4, r4, 1                /* Decrement the counter */
         bgt    r4, r0, LOOP             /* Loop again if not finished */
         stw    r5, DOT_PRODUCT(r0)      /* Store the result in memory */
STOP:    br     STOP
.org     0xdf0
.word    6                               /* Specify the number of elements */
.org     0xe00
.word    5, 3, −6, 19, 8, 12             /* Specify the elements of vector A */
.org     0xf00
.word    2, 14, −3, 2, −5, 36            /* Specify the elements of vector B */
```

Factorial

```
fact:     addi $1, $0, 1          # initialize reg. 1 to 1
          beq $4, $0, return1     # if (n == 0) then goto return1:
          bne $4, $1, continue    # if (n != 1) then goto continue:
return1:          addi $2, $1, 0          # assign result = 1
          jr $31                  # return

continue: addi $29, $29, -12      # allocate stack space for fp, ra, n
          sw $30, 8 ($29)         # save frame pointer
          sw $31, 4 ($29)         # save return address
          addi $30, $29, 8        # update frame pointer
          sw $4, 0 ($29)          # save n
          addi $4, $4, -1         # make n-1
          jal fact                # recursive call to fact(n-1)
          lw $4, 0 ($29)          # restore n
          mult $2, $4             # fact (n-1) * n
          mflo $2                 # put product in result reg.
          lw $31, 4 ($29)         # restore return address
          lw $30, 8 ($29)         # restore frame pointer
          addi $29, $29, 12       # restore stack pointer
          jr $31                  # return
```

Instruction Signal Paths

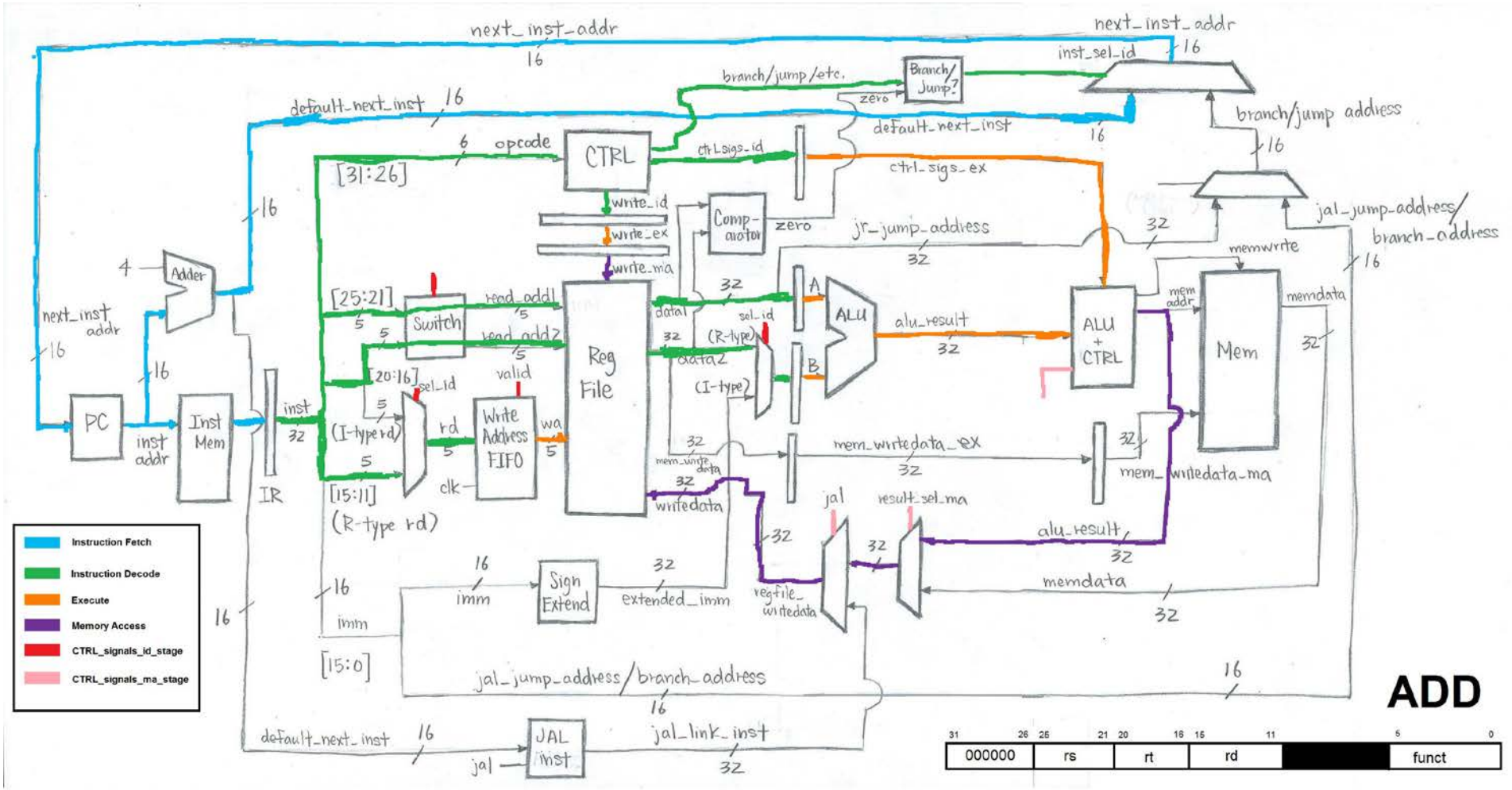


Figure 7: ADD instruction signal path

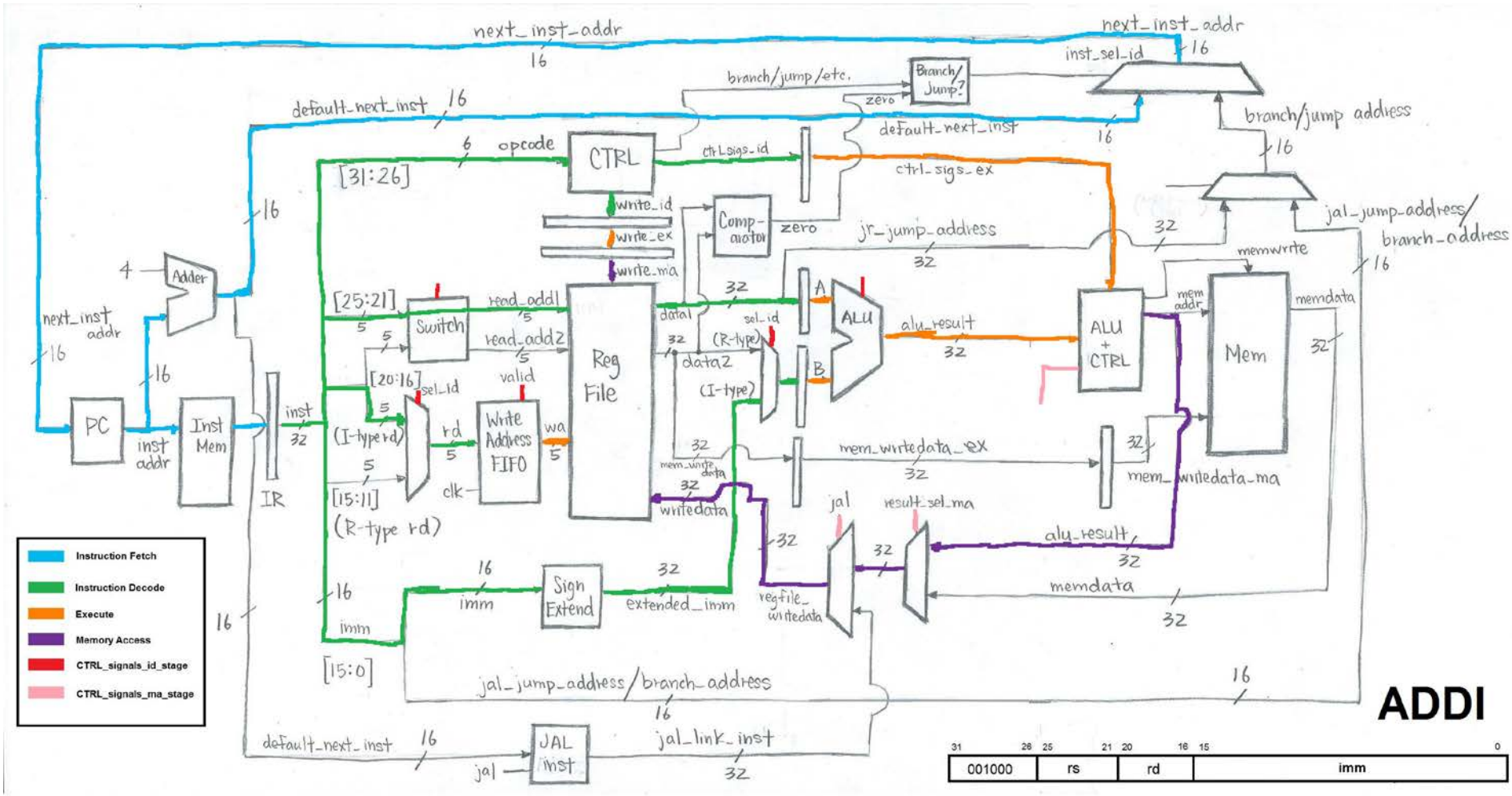


Figure 8: ADDI instruction signal path

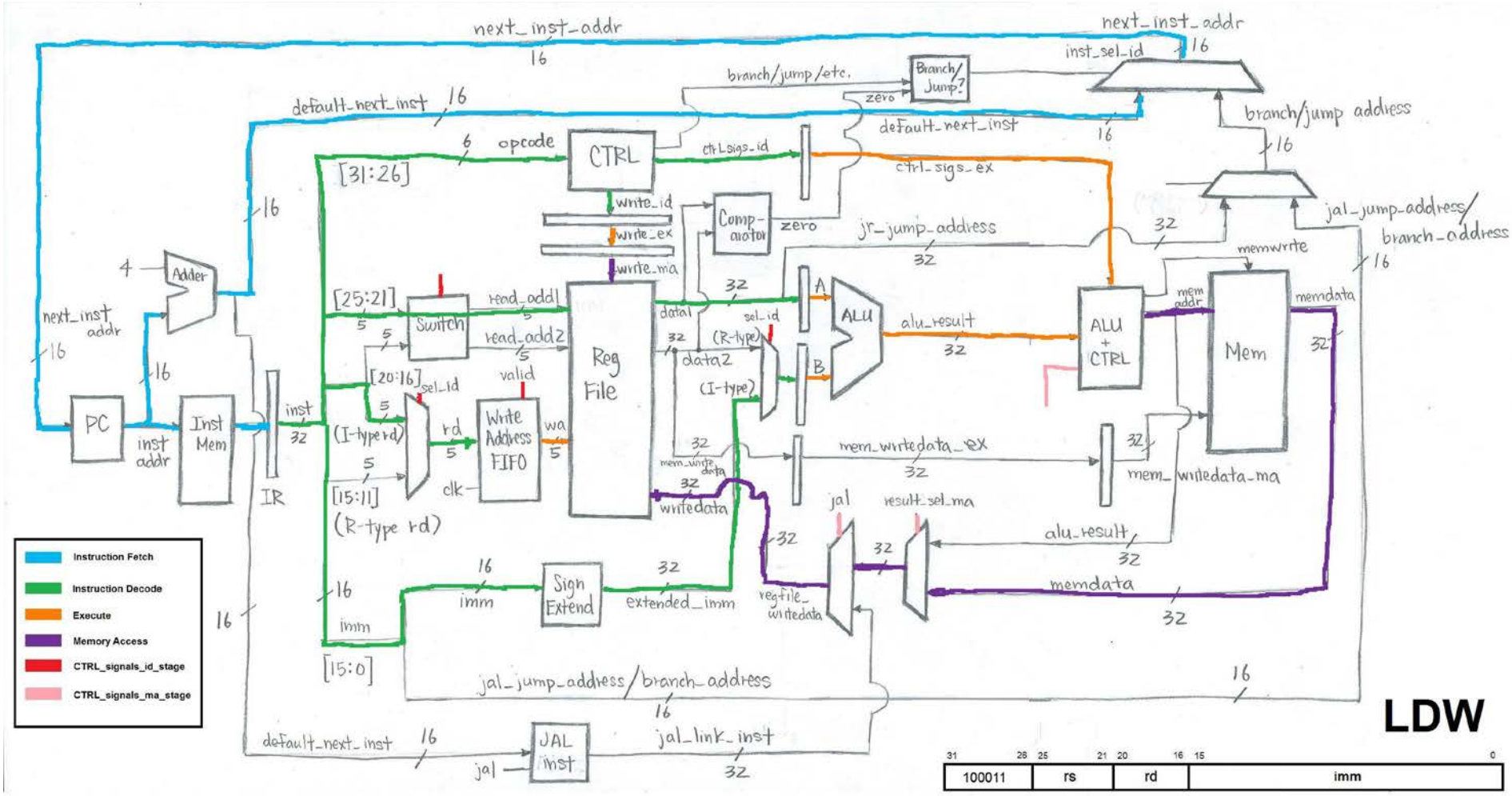


Figure 9: LDW instruction signal path

- While not shown, the Register File and Memory both write the input data into the input address every clock cycle if their write signal is high. By the next clock edge after the memory access cycle, it can be seen that the writedata, write signal, and write address of the register file are ready.

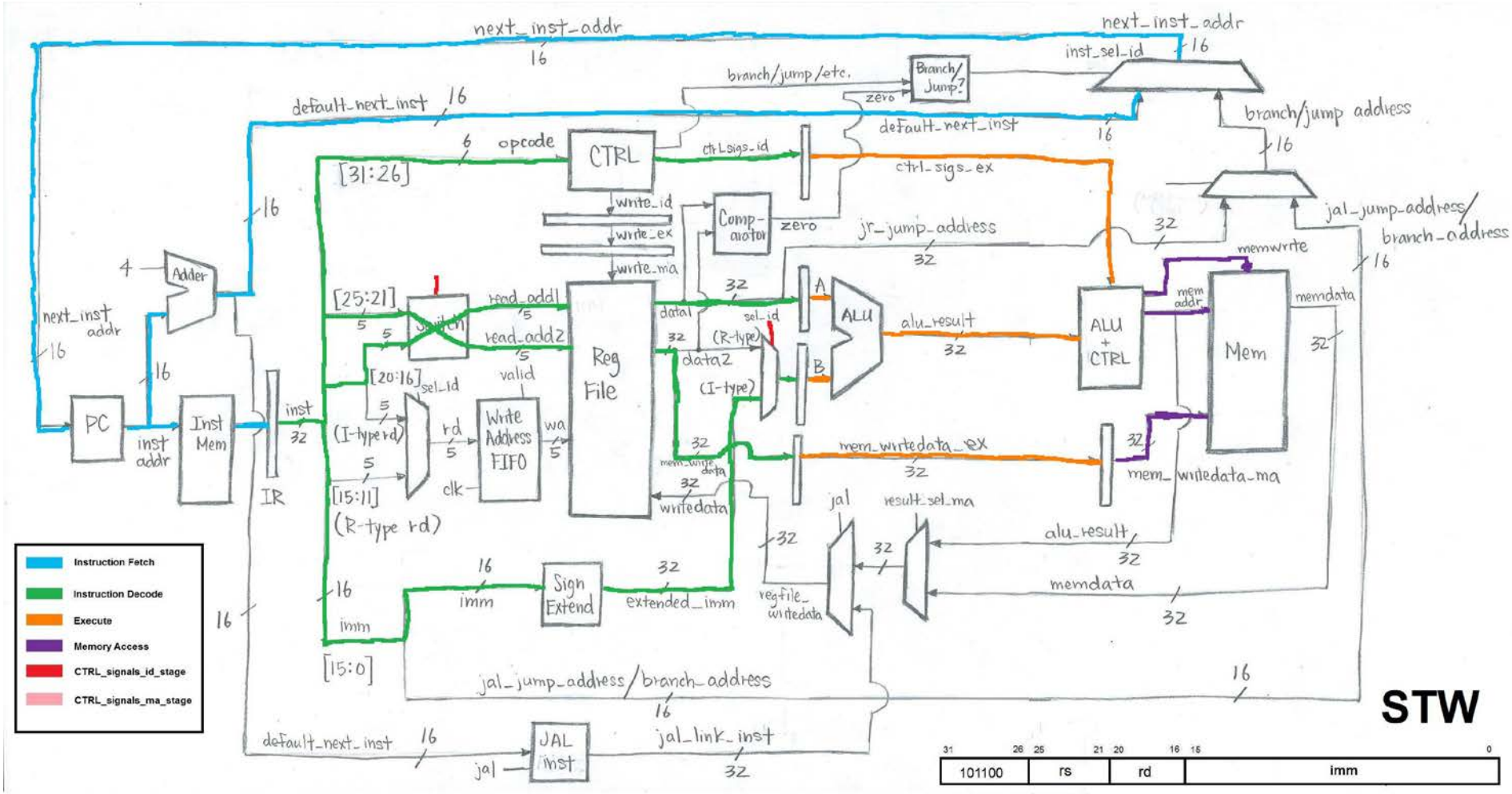


Figure 10: STW instruction signal path

- Here, the contents of the “rs” register are output from data2out of the register file and towards memory_writedata. The signal switch was implemented to keep all instructions consistent in which bits represented “rs”
- “rd” is sent to the ALU with the immediate offset value for the destination memory address

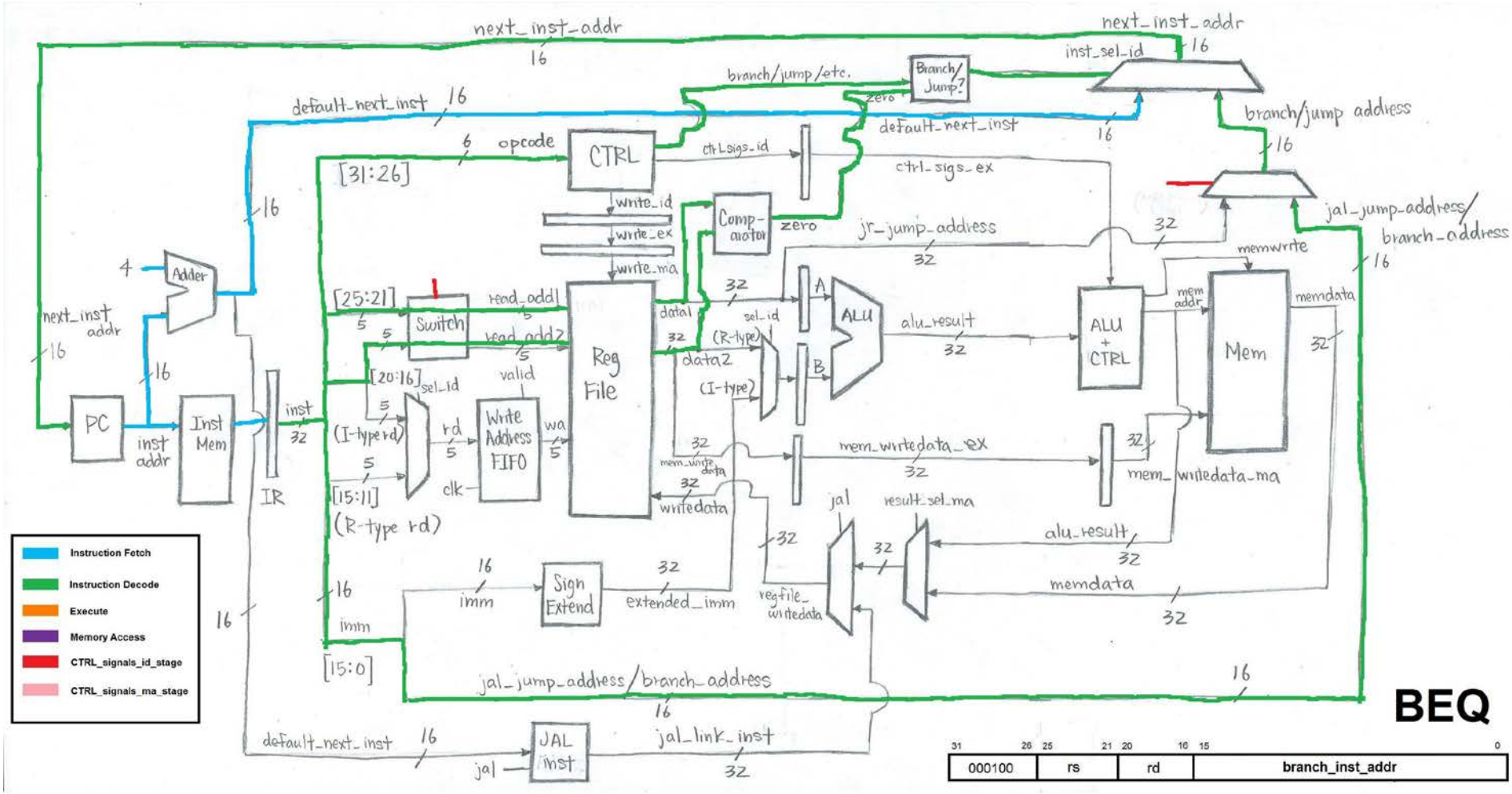


Figure 11: BEQ instruction signal path

- When branching or jumping, we want the next instruction as soon as possible, so modules were added to skip all stage-transition flip flops
- As mentioned before, a dead instruction is needed after every branch/jump instruction because it takes 1 clock cycle to set the branch/jump address.

  Ex. Current instruction address is 12. Since the branch opcode hasn't gone past the IR flip flop, the next instruction address is set at 16 instead of branch address 32. After the instruction is clocked through, the address going into the PC is set to 32. However, instruction 16 has entered the pipelined and will be executed.

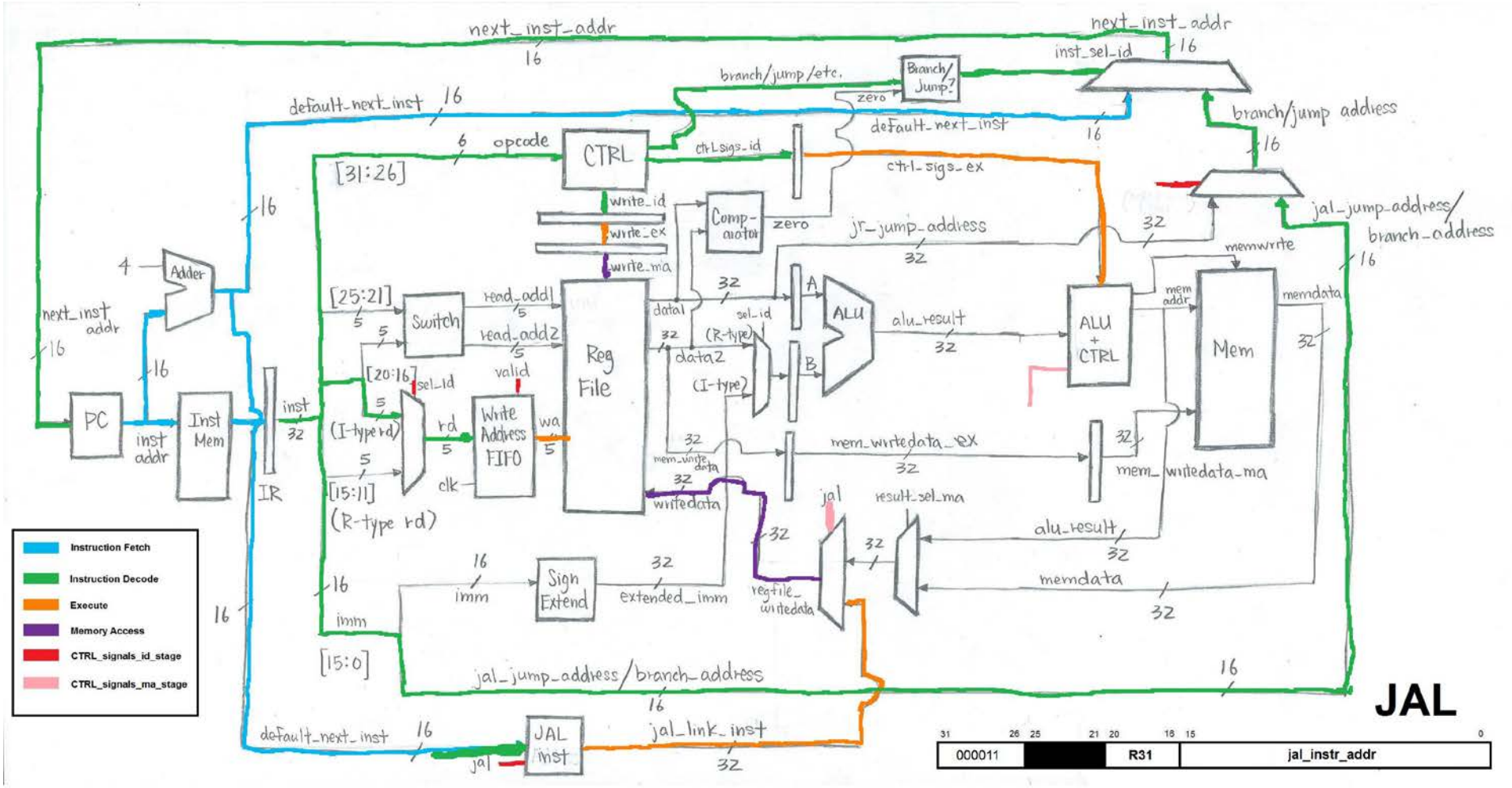


Figure 12: JAL instruction signal path

- The JAL_inst register at the very bottom captures the return address every time the jal signal goes high from the CTRL module. The signal gets captured during the decode stage because that is when the CTRL module gets the opcode and sends out the jal signal.

  Ex. Current instruction address is 12. The address into the JAL_inst register is at 16 but since the branch opcode hasn't gone past the IR flip flop, the register doesn't save that address. After the jal instruction is clocked through the IR register, the address going into the JAL_inst register is 20. Therefore, the return instruction address will be written to be 20, and instruction 16 needs to be a dead instruction.

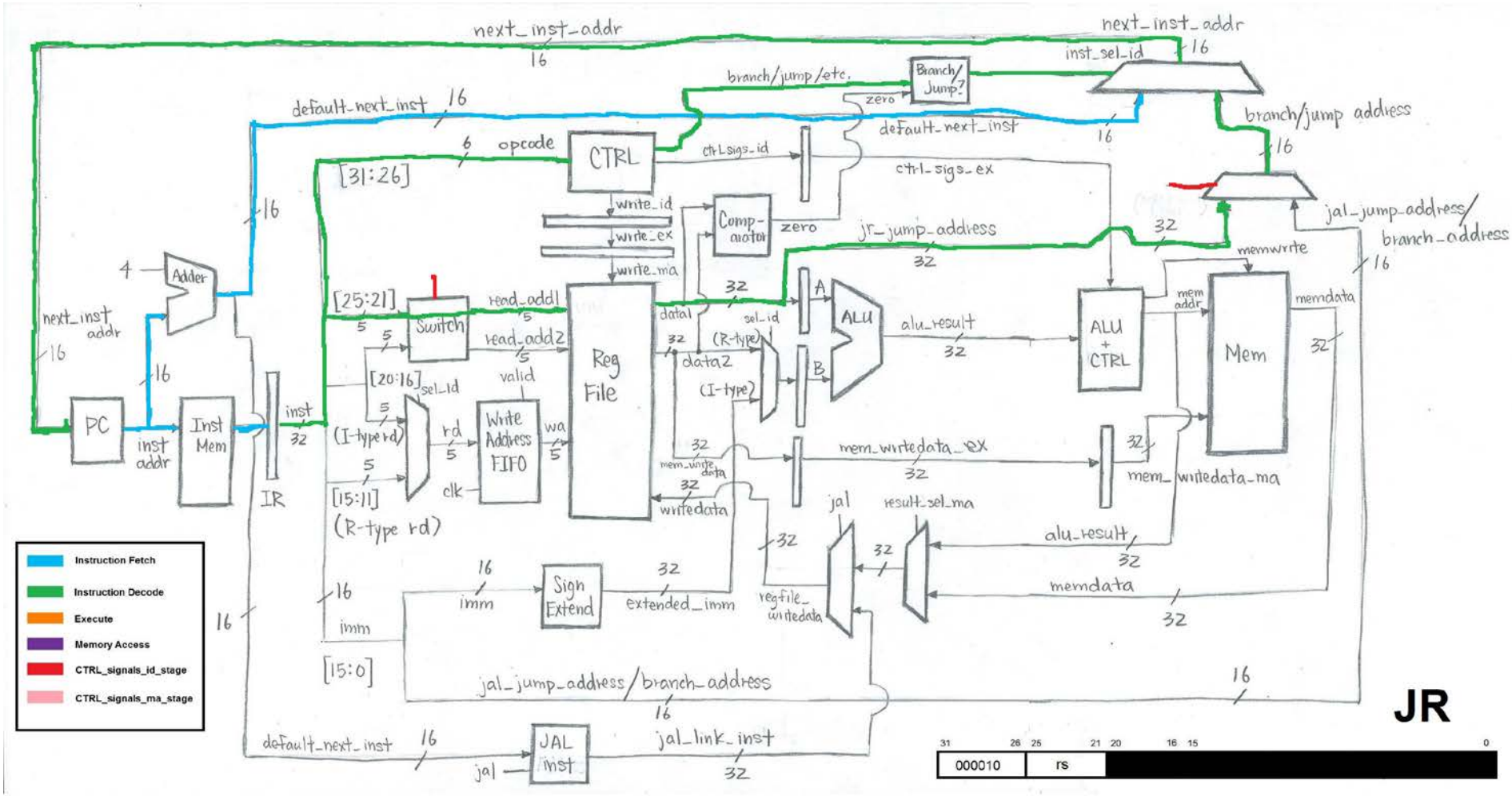
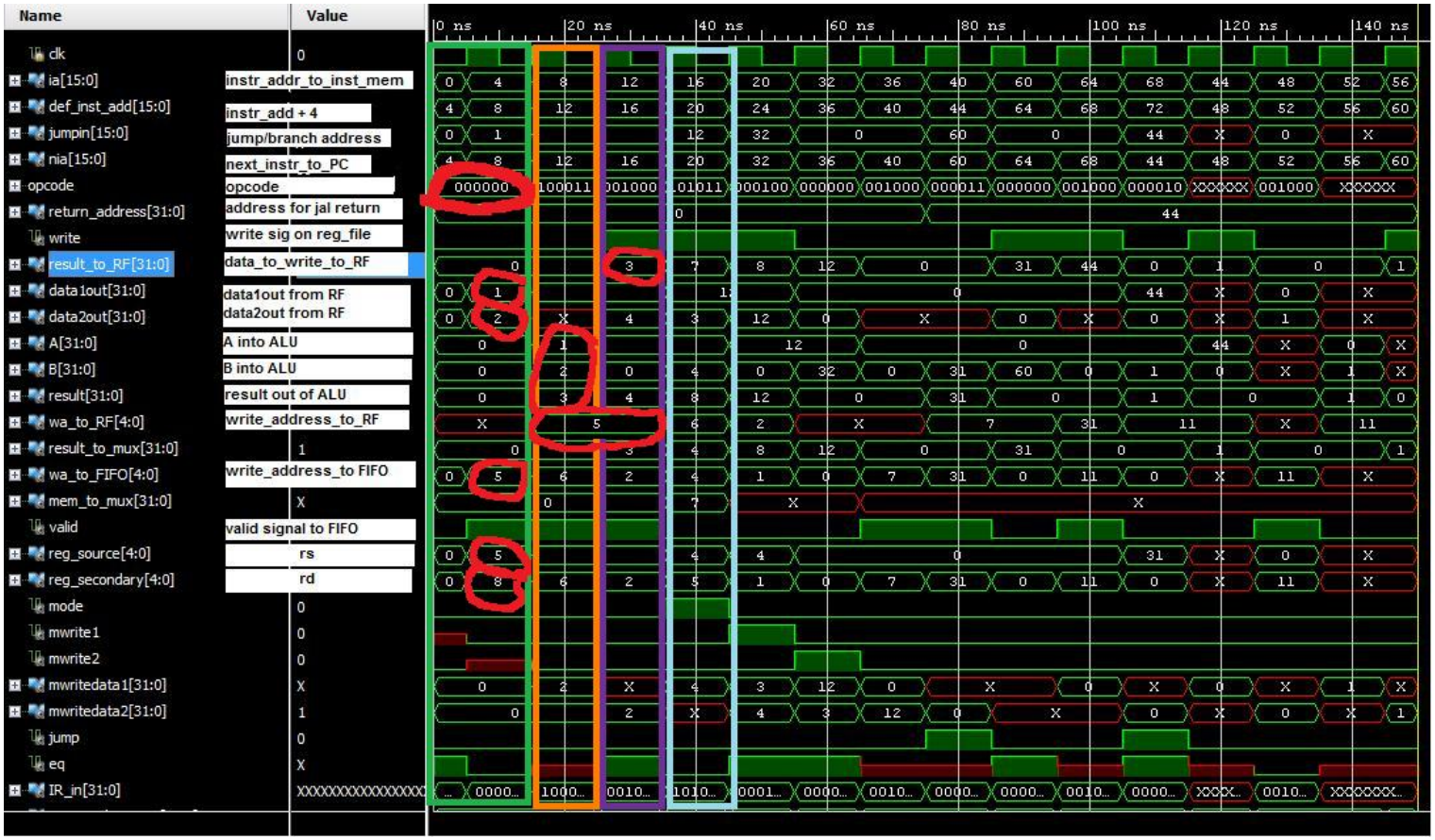


Figure 13: JR instruction signal path

Waveforms of Pipelined Instructions



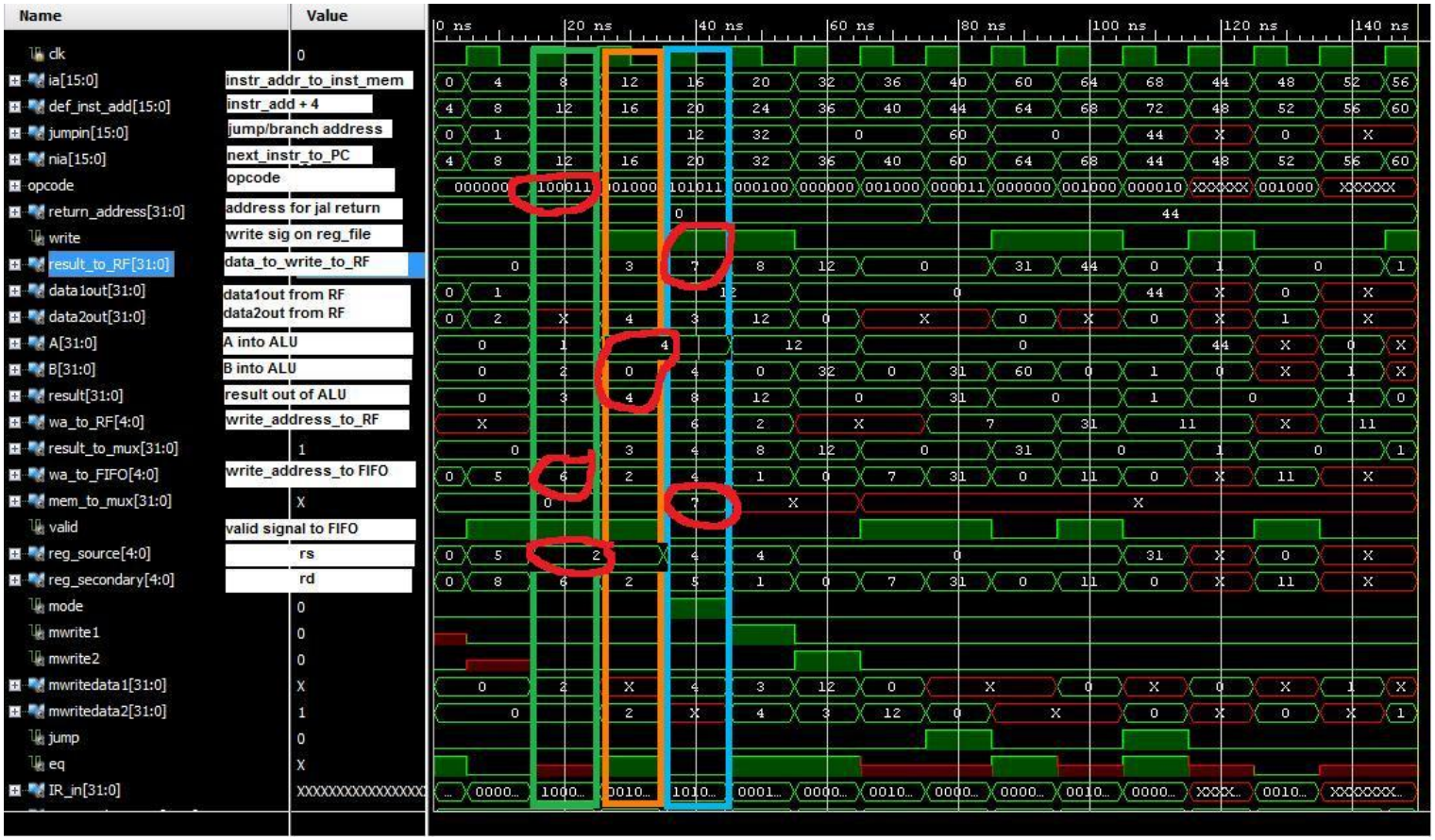
ADD

ADD R5, R5, R8
opcode = 000000

R5 = 1
R8 = 2

RS = R5
RT = R8
RD = R5

Figure 14: ADD instruction waveform



LDW

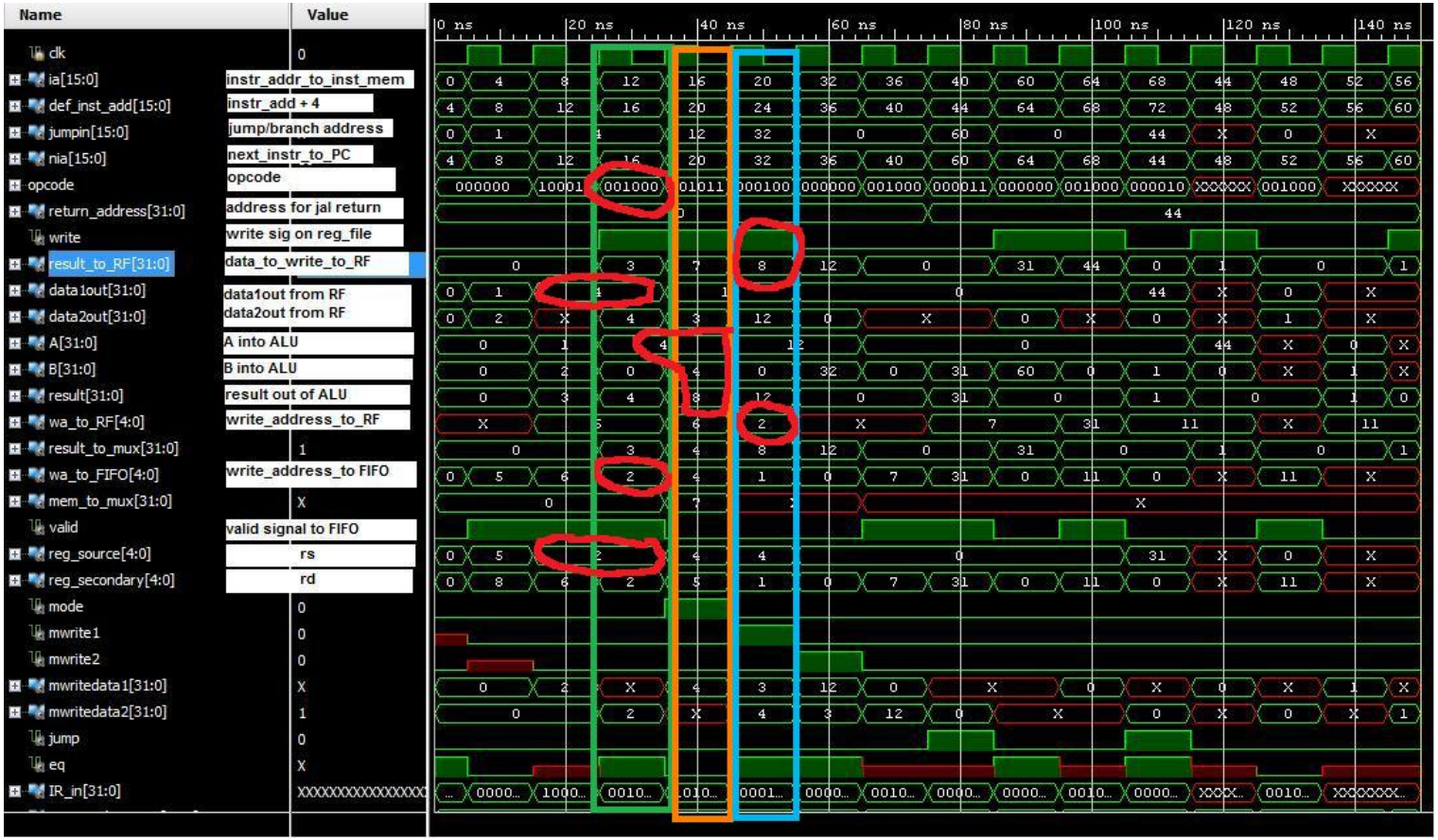
opcode = 100011

LDW R6, 0(R2)

R6 = empty
R2 = 4

M4 = 7

Figure 15: LDW instruction waveform



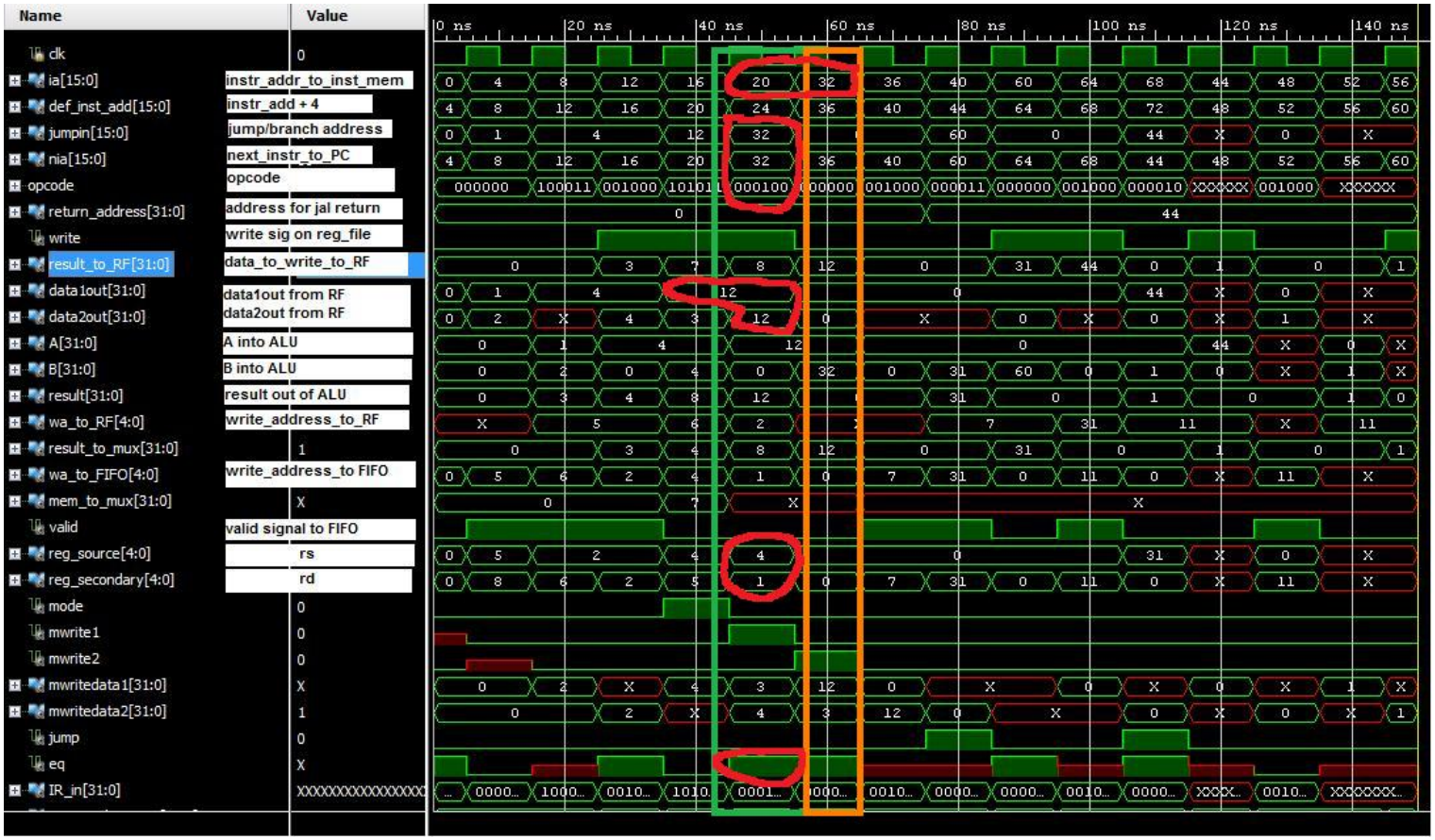
ADDI

opcode = 001000

ADDI R2, R2, #4

R2 = 4

Figure 16: ADDI instruction waveform



BEQ

opcode = 000100

BEQ R4, R1, #32

R1 = 12
R4 = 12

Figure 17: BEQ instruction waveform

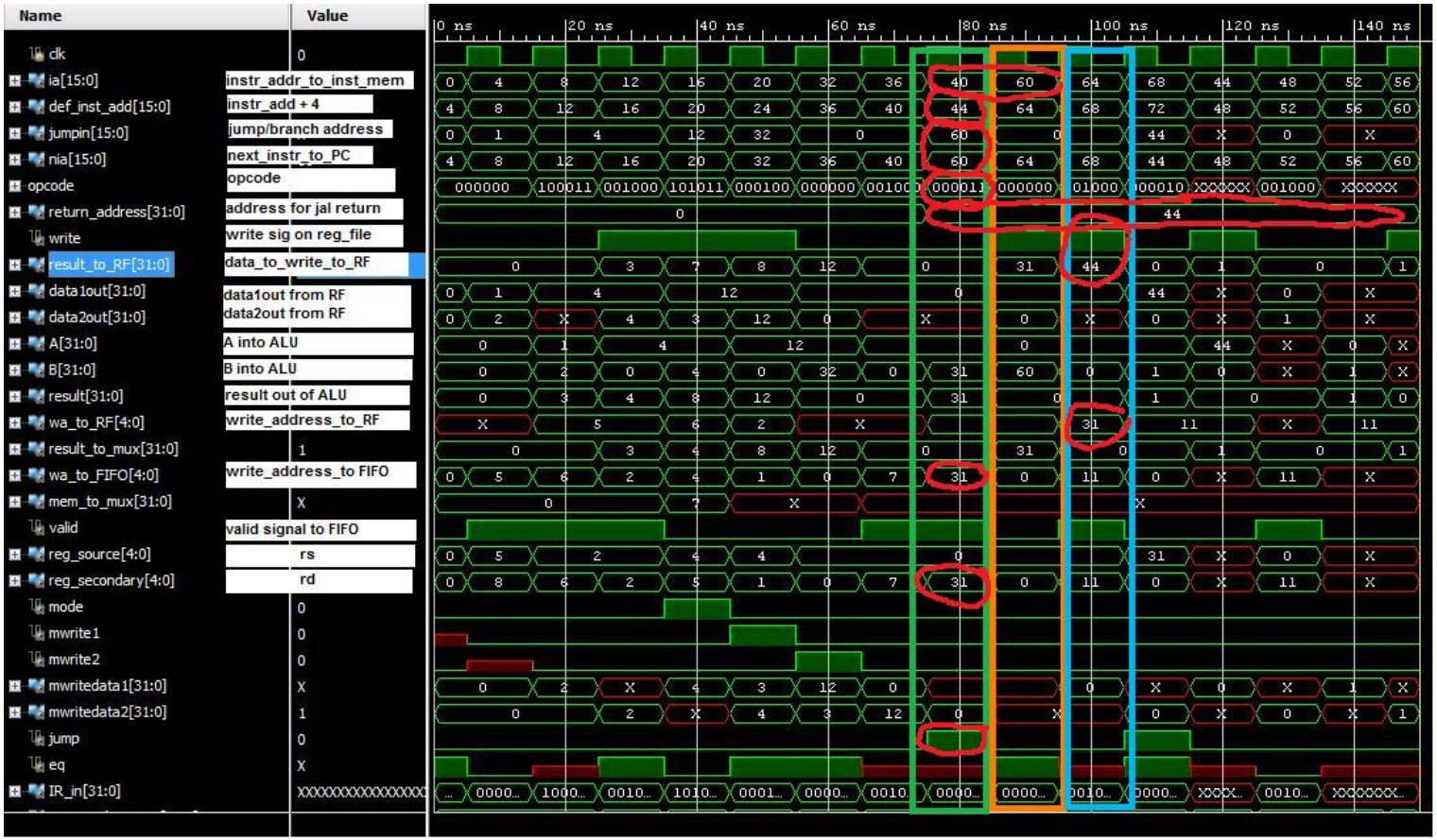


Figure 18: JAL instruction waveform

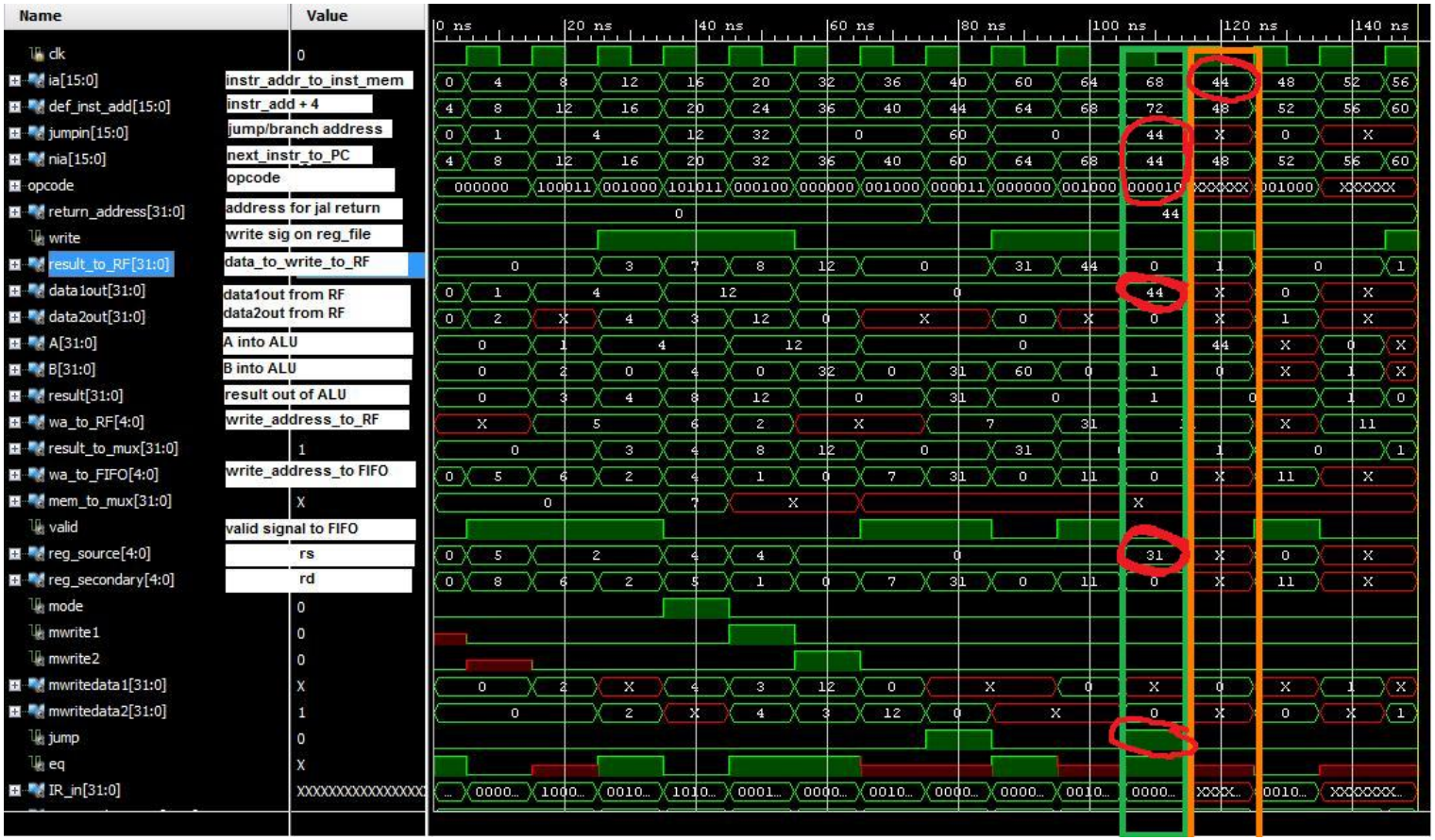


Figure 19: JR instruction waveform