論理回路入門

SpringCPU 設計

李　亜民

2026 年春休み

春休みの自由練習

SpringCPU

設計

SpringCPU 命令の意味

```
 0. nop                  # no operation
 1. add   rd, rs1, rs2   # rd <- rs1  +  rs2, add
 2. sub   rd, rs1, rs2   # rd <- rs1  -  rs2, subtract
 3. and   rd, rs1, rs2   # rd <- rs1  &  rs2, logic and
 4. or    rd, rs1, rs2   # rd <- rs1  |  rs2, logic or
 5. xor   rd, rs1, rs2   # rd <- rs1  ^  rs2, logic xor
 6. sll   rd, rs1, rs2   # rd <- rs1 <<  rs2, shift left  logical
 7. srl   rd, rs1, rs2   # rd <- rs1 >>  rs2, shift right logical
 8. sra   rd, rs1, rs2   # rd <- rs1 >>> rs2, shift right arithmetic
 9. slt   rd, rs1, rs2   # rd <- rs1  <  rs2, set on less than
10. addi  rd, rs1, imm   # rd <- rs1  +  imm, add immediate, imm:   signed
11. andi  rd, rs1, imm   # rd <- rs1  &  imm, and immediate, imm: unsigned
12. ori   rd, rs1, imm   # rd <- rs1  |  imm, or  immediate, imm: unsigned
13. xori  rd, rs1, imm   # rd <- rs1  ^  imm, xor immediate, imm: unsigned
14. slli  rd, rs1, imm   # rd <- rs1 <<  imm, sll immediate
15. srli  rd, rs1, imm   # rd <- rs1 >>  imm, srl immediate
16. srai  rd, rs1, imm   # rd <- rs1 >>> imm, sra immediate
17. slti  rd, rs1, imm   # rd <- rs1  <  imm, slt immediate, imm:   signed
18. load  rd,  imm(rs1) # rd <- memory[rs1+imm],  imm: unsigned, load from memory
19. store rs2, imm(rs1) # memory[rs1+imm] <- rs2, imm: unsigned, store to  memory
20. bz    rs1, target   # if (rs1==0) pc <- pc + offset, branch on zero
21. bnz   rs1, target   # if (rs1!=0) pc <- pc + offset, branch on not zero
22. li    rd,  imm8     # load 8-bit immediate
```

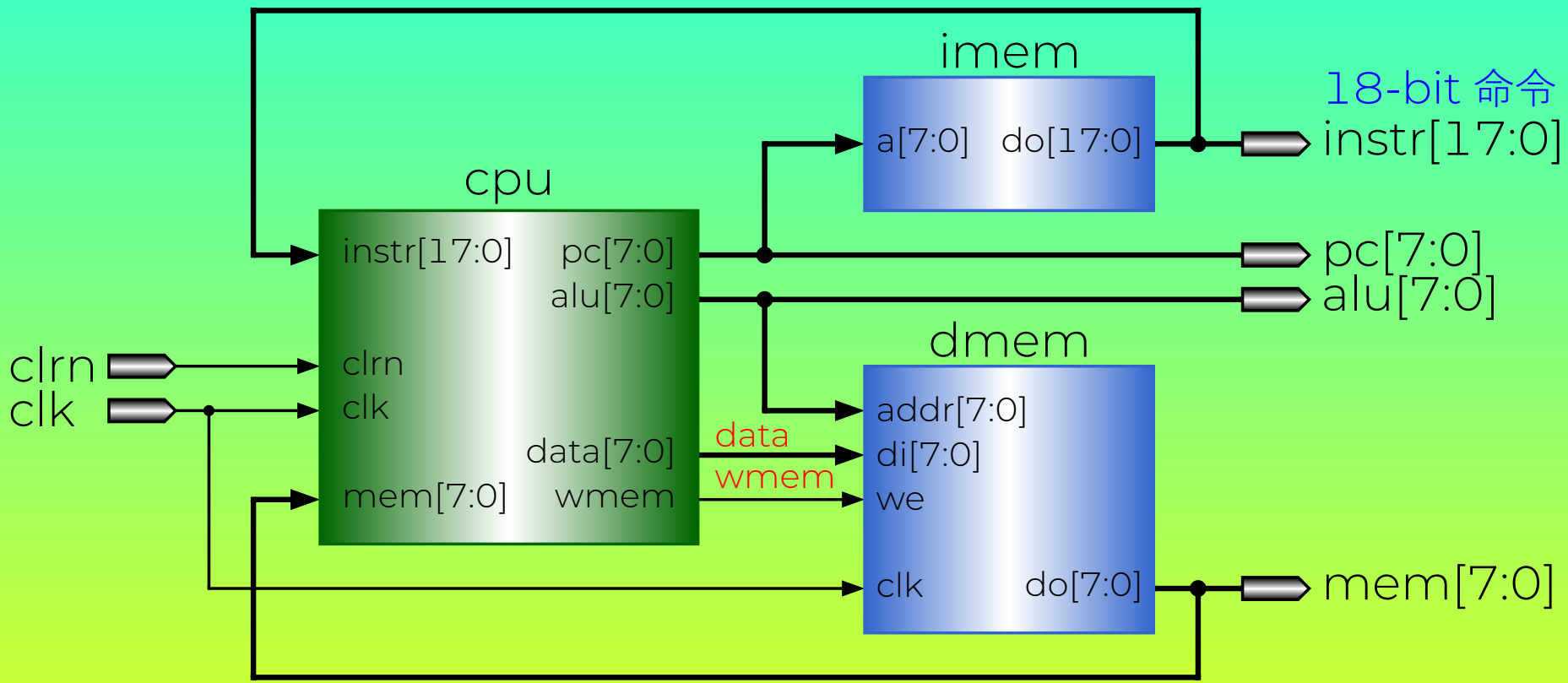
SpringCPU 命令のフォーマット

```
 0. nop                  # 000000 0000 0000 0000 # instruction:    18 bits
 1. add   rd, rs1, rs2   # 000001 rs1, rd,  rs2  # opcode:          6 bits
 2. sub   rd, rs1, rs2   # 000010 rs1, rd,  rs2  # register number: 4 bits
 3. and   rd, rs1, rs2   # 000011 rs1, rd,  rs2  # program counter: 8 bits
 4. or    rd, rs1, rs2   # 000100 rs1, rd,  rs2  # data:            8 bits
 5. xor   rd, rs1, rs2   # 000101 rs1, rd,  rs2  #
 6. sll   rd, rs1, rs2   # 000110 rs1, rd,  rs2  # registers:           16
 7. srl   rd, rs1, rs2   # 000111 rs1, rd,  rs2  # instruction memory: 256
 8. sra   rd, rs1, rs2   # 001000 rs1, rd,  rs2  # data memory:        256
 9. slt   rd, rs1, rs2   # 001001 rs1, rd,  rs2  #
10. addi  rd, rs1, imm   # 001010 rs1, rd,  imm  #
11. andi  rd, rs1, imm   # 001011 rs1, rd,  imm  #
12. ori   rd, rs1, imm   # 001100 rs1, rd,  imm  #
13. xori  rd, rs1, imm   # 001101 rs1, rd,  imm  #
14. slli  rd, rs1, imm   # 001110 rs1, rd,  imm  #
15. srli  rd, rs1, imm   # 001111 rs1, rd,  imm  #
16. srai  rd, rs1, imm   # 010000 rs1, rd,  imm  #
17. slti  rd, rs1, imm   # 010001 rs1, rd,  imm  #
18. load  rd,  imm(rs1)  # 010010 rs1, rd,  imm  #
19. store rs2, imm(rs1)  # 010011 rs1, imm, rs2  #
20. bz    rs1, target    # 010100 rs1, imm  imm  # offset = imm_imm
21. bnz   rs1, target    # 010101 rs1, imm  imm  # offset = imm_imm
22. li    rd,  imm8      # 010110 imm, rd,  imm  # imm8   = imm_imm
```

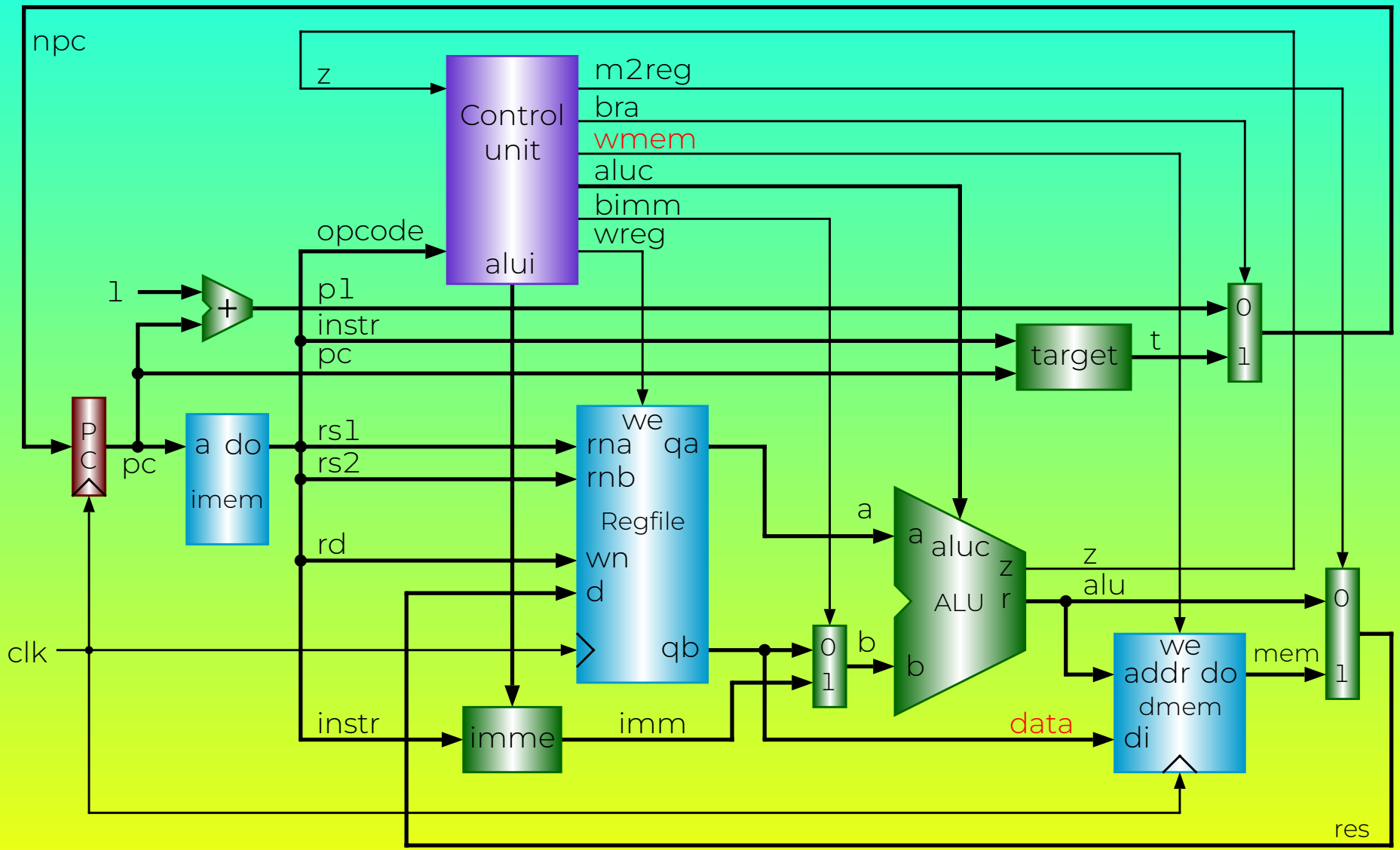
SpringCPU 命令の ALU 操作

```
 0. nop                  # 000000 0000 0000 0000 # no operation                aluc
 1. add   rd, rs1, rs2   # 000001 rs1, rd,  rs2  # ADD ..................... 0000
 2. sub   rd, rs1, rs2   # 000010 rs1, rd,  rs2  # SUB ..................... 0001
 3. and   rd, rs1, rs2   # 000011 rs1, rd,  rs2  # AND ..................... 0010
 4. or    rd, rs1, rs2   # 000100 rs1, rd,  rs2  # OR  ..................... 0011
 5. xor   rd, rs1, rs2   # 000101 rs1, rd,  rs2  # XOR .................... 0100
 6. sll   rd, rs1, rs2   # 000110 rs1, rd,  rs2  # SLL ..................... 1000
 7. srl   rd, rs1, rs2   # 000111 rs1, rd,  rs2  # SRL ..................... 1001
 8. sra   rd, rs1, rs2   # 001000 rs1, rd,  rs2  # SRA ..................... 1011
 9. slt   rd, rs1, rs2   # 001001 rs1, rd,  rs2  # SLT ..................... 0101
10. addi  rd, rs1, imm   # 001010 rs1, rd,  imm  # same as ADD
11. andi  rd, rs1, imm   # 001011 rs1, rd,  imm  # same as AND
12. ori   rd, rs1, imm   # 001100 rs1, rd,  imm  # same as OR
13. xori  rd, rs1, imm   # 001101 rs1, rd,  imm  # same as XOR
14. slli  rd, rs1, imm   # 001110 rs1, rd,  imm  # same as SLL
15. srli  rd, rs1, imm   # 001111 rs1, rd,  imm  # same as SRL
16. srai  rd, rs1, imm   # 010000 rs1, rd,  imm  # same as SRA
17. slti  rd, rs1, imm   # 010001 rs1, rd,  imm  # same as SLT
18. load  rd,  imm(rs1)  # 010010 rs1, rd,  imm  # same as ADD
19. store rs2, imm(rs1)  # 010011 rs1, imm, rs2  # same as ADD
20. bz    rs1, target    # 010100 rs1, imm  imm  # same as ADD addi 0, check z
21. bnz   rs1, target    # 010101 rs1, imm  imm  # same as ADD addi 0, check z
22. li    rd,  imm8      # 010110 imm, rd,  imm  # LI ..................... 1111
```

SpringCPU + メモリ



SpringCPU + メモリ



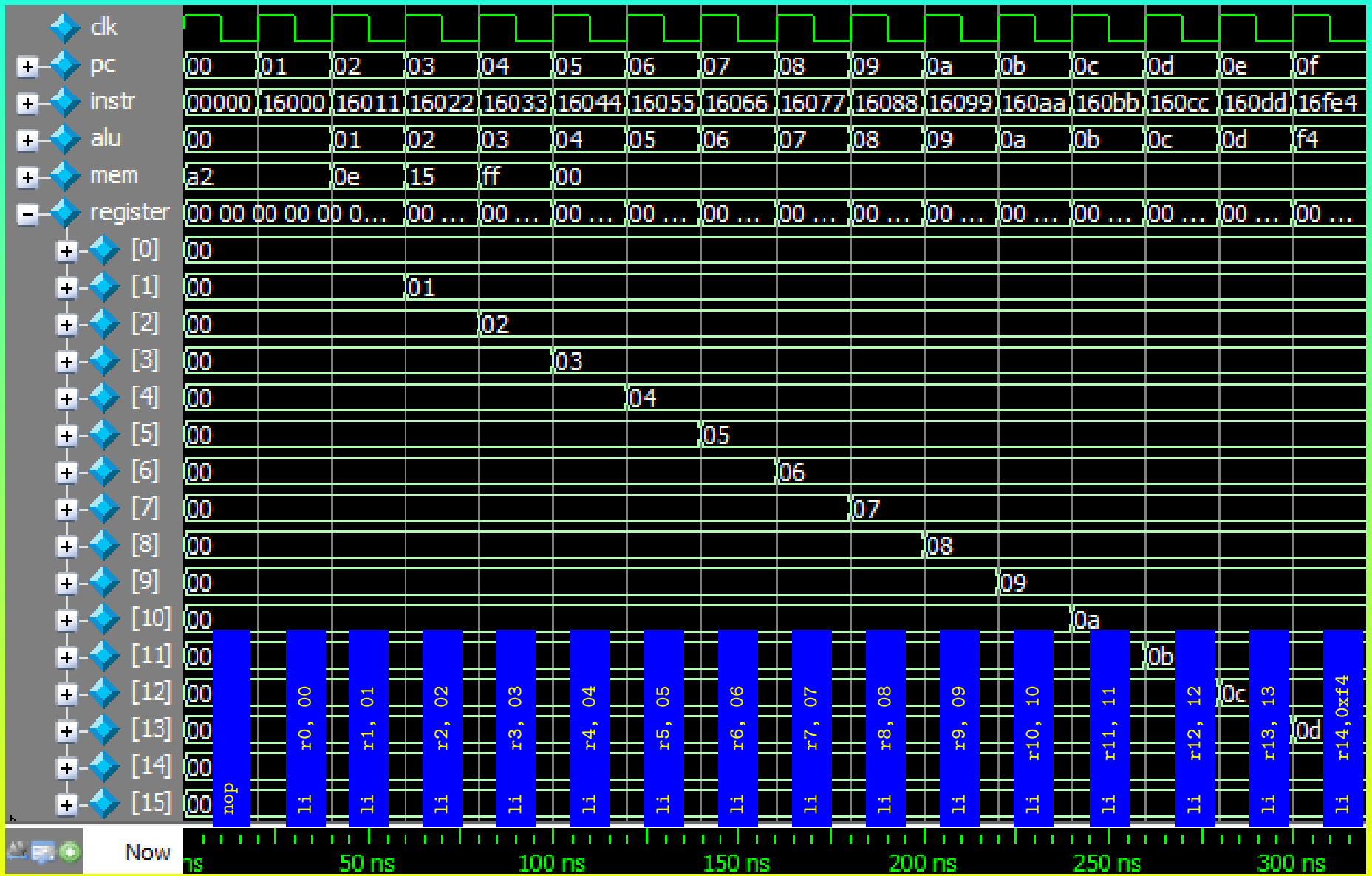
Test Code (imem.v)

```
‘timescale 1ns/1ns
module imem (a, do);
    input   [7:0] a;
    output [17:0] do;
    wire   [17:0] rom [0:255];
    assign do = rom[a];
    //                      opcode imm, rd,  imm
    assign rom[8’h00] = 18’b000000_0000_0000_0000; // main: nop
    assign rom[8’h01] = 18’b010110_0000_0000_0000; //       li    r0,  00        #  0
    assign rom[8’h02] = 18’b010110_0000_0001_0001; //       li    r1,  01        #  1
    assign rom[8’h03] = 18’b010110_0000_0010_0010; //       li    r2,  02        #  2
    assign rom[8’h04] = 18’b010110_0000_0011_0011; //       li    r3,  03        #  3
    assign rom[8’h05] = 18’b010110_0000_0100_0100; //       li    r4,  04        #  4
    assign rom[8’h06] = 18’b010110_0000_0101_0101; //       li    r5,  05        #  5
    assign rom[8’h07] = 18’b010110_0000_0110_0110; //       li    r6,  06        #  6
    assign rom[8’h08] = 18’b010110_0000_0111_0111; //       li    r7,  07        #  7
    assign rom[8’h09] = 18’b010110_0000_1000_1000; //       li    r8,  08        #  8
    assign rom[8’h0a] = 18’b010110_0000_1001_1001; //       li    r9,  09        #  9
    assign rom[8’h0b] = 18’b010110_0000_1010_1010; //       li   r10,  10        # 10
    assign rom[8’h0c] = 18’b010110_0000_1011_1011; //       li   r11,  11        # 11
    assign rom[8’h0d] = 18’b010110_0000_1100_1100; //       li   r12,  12        # 12
    assign rom[8’h0e] = 18’b010110_0000_1101_1101; //       li   r13,  13        # 13
    assign rom[8’h0f] = 18’b010110_1111_1110_0100; //       li   r14, 0xf4       # 0xf4
    assign rom[8’h10] = 18’b010110_1111_1111_0101; //       li   r15, 0xf5       # 0xf5
    //                      opcode rs1, rd,  rs2
    assign rom[8’h11] = 18’b000001_0001_0000_0010; //       add   r0,  r1,  r2   #  3 = 1 + 2
    assign rom[8’h12] = 18’b000010_0011_0001_0100; //       sub   r1,  r3,  r4   # -1 = 3 - 4
    assign rom[8’h13] = 18’b000011_0101_0010_0110; //       and   r2,  r5,  r6   #  4 = 5 & 6
    assign rom[8’h14] = 18’b000100_0111_0011_1000; //       or    r3,  r7,  r8   # 15 = 7 | 8
    assign rom[8’h15] = 18’b000101_1001_0100_1010; //       xor   r4,  r9, r10   #  3 = 9 ^ 10
    assign rom[8’h16] = 18’b000110_1011_0101_0010; //       sll   r5, r11,  r2   # 0xb0 = 0x0b << 4
```

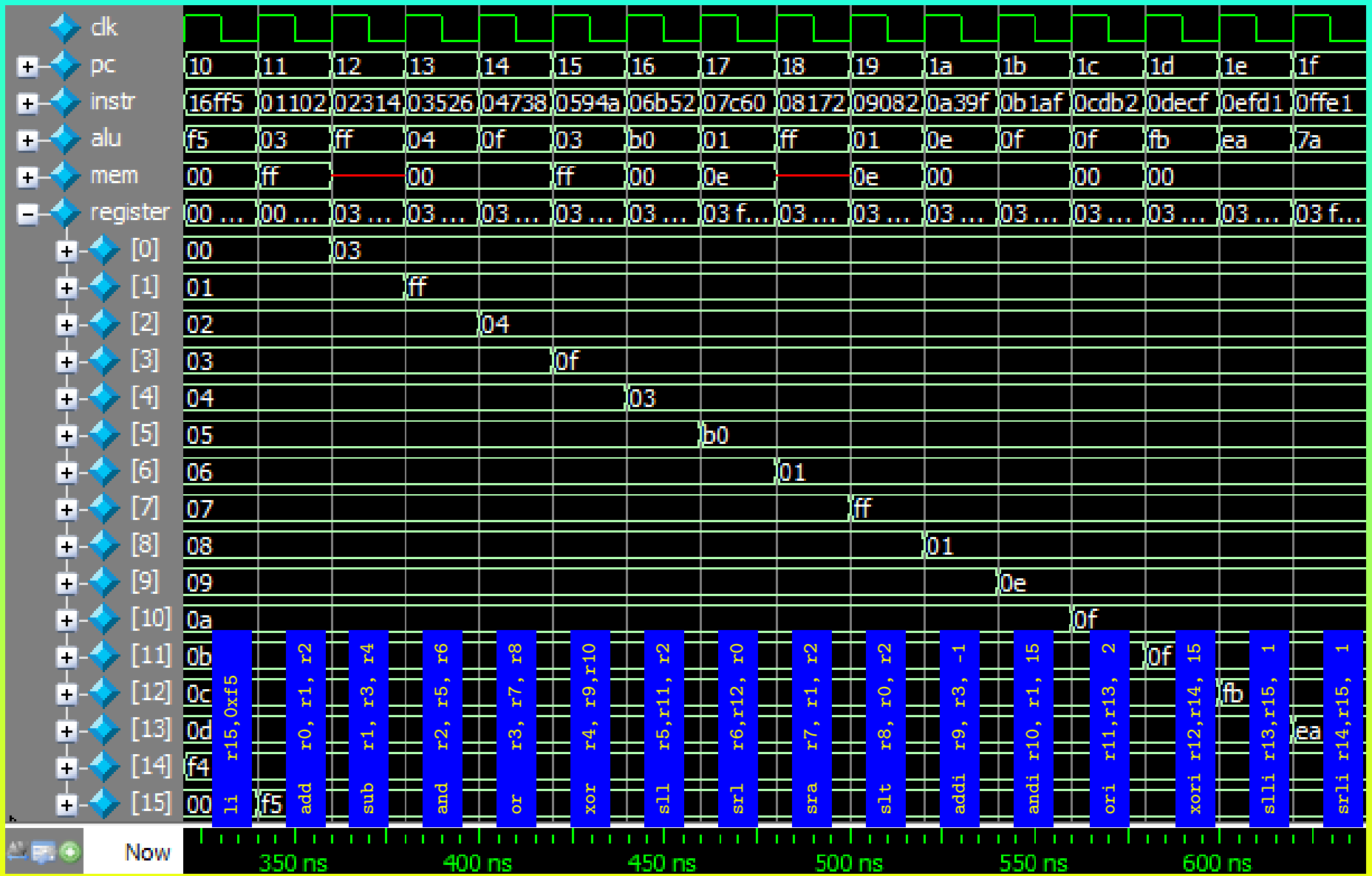
Test Code (imem.v)

```
    assign rom[8'h17] = 18'b000111_1100_0110_0000; //       srl   r6, r12,  r0   # 1 = 0x0c >> 3
    assign rom[8'h18] = 18'b001000_0001_0111_0010; //       sra   r7,  r1,  r2   # 0xff = 0xff >>> 4
    assign rom[8'h19] = 18'b001001_0000_1000_0010; //       slt   r8,  r0,  r2   # 1 = 3 < 4
    //                      opcode rs1, rd,  imm
    assign rom[8'h1a] = 18'b001010_0011_1001_1111; //       addi  r9,  r3,  -1   # 14 = 15 + (-1)
    assign rom[8'h1b] = 18'b001011_0001_1010_1111; //       andi r10,  r1,  15   # 0x0f = 0xff & 0x0f
    assign rom[8'h1c] = 18'b001100_1101_1011_0010; //       ori  r11, r13,   2   # 15 = 13 | 2
    assign rom[8'h1d] = 18'b001101_1110_1100_1111; //       xori r12, r14,  15   # 0xfb = 0xf4 ^ 0x0f
    assign rom[8'h1e] = 18'b001110_1111_1101_0001; //       slli r13, r15,   1   # 0xea = 0xf5 <<  1
    assign rom[8'h1f] = 18'b001111_1111_1110_0001; //       srli r14, r15,   1   # 0x7a = 0xf5 >>  1
    assign rom[8'h20] = 18'b010000_1111_1111_0001; //       srai r15, r15,   1   # 0xfa = 0xf5 >>> 1
    assign rom[8'h21] = 18'b010001_0001_0000_0001; //       slti  r0,  r1,   1   # 1 = -1 < 1
    //                      opcode imm, rd,  imm
    assign rom[8'h22] = 18'b010110_0000_0000_0100; // sum:  li    r0,   4         # count = 4
    assign rom[8'h23] = 18'b010110_0000_0001_0000; //       li    r1,   0         # sum = 0
    assign rom[8'h24] = 18'b010110_0000_0010_0000; //       li    r2,   0         # address = 0
    //                      opcode rs1, rd,  imm
    assign rom[8'h25] = 18'b010010_0010_0011_0000; // loop: load  r3, 0(r2)       # load data from memory
    assign rom[8'h26] = 18'b001010_0010_0010_0001; //       addi  r2,  r2,   1   # address = address + 1
    assign rom[8'h27] = 18'b001010_0000_0000_1111; //       addi  r0,  r0,  -1   # count = count - 1
    //                      opcode rs1, rd,  rs2
    assign rom[8'h28] = 18'b000001_0001_0001_0011; //       add   r1,  r1,  r3   # sum = sum + data
    //                      opcode rs1, imm  imm
    assign rom[8'h29] = 18'b010101_0000_1111_1100; //       bnz   r0, loop       # goto loop if count != 0
    //                      opcode rs1, imm, rs2
    assign rom[8'h2a] = 18'b010011_0001_0000_0010; //       store r1, 0(r2)      # store data to memory
    //                      opcode rs1, imm  imm
    assign rom[8'h2b] = 18'b010100_0000_0000_0010; // back: bz    r0, next       # jump to next (r0 == 0)
    assign rom[8'h2c] = 18'b000000_0000_0000_0000; //       nop                  # nop
    assign rom[8'h2d] = 18'b010100_0000_1111_1110; // next: bz    r0, back       # jump to back, dead loop
endmodule
```

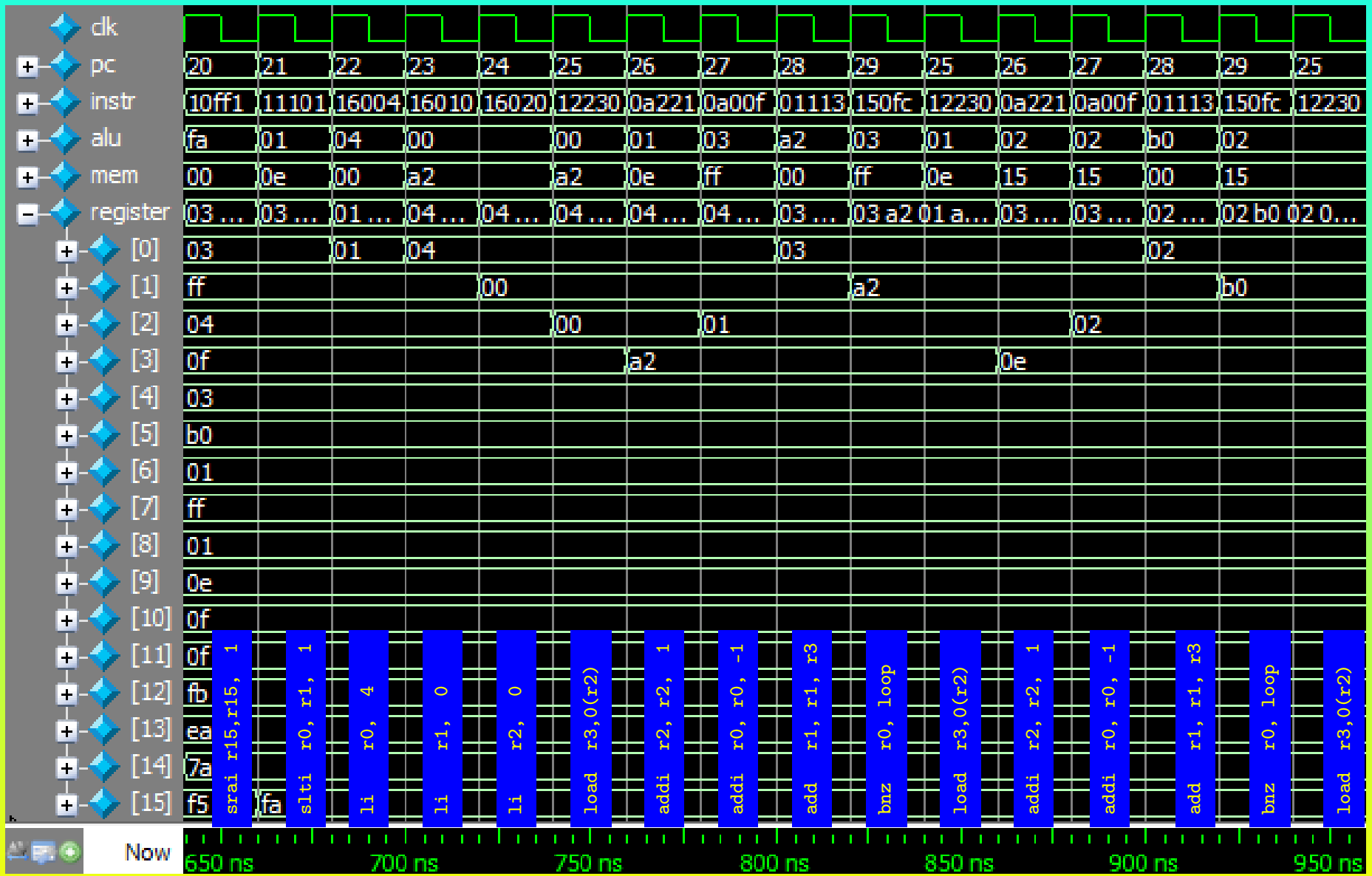
波形1


clk
pc
instr
alu
mem
register
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
Now
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns

波形2


clk
pc
instr
alu
mem
register
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
Now
350 ns
400 ns
450 ns
500 ns
550 ns
600 ns
li r15,0xf5
add r0, r1, r2
sub r1, r3, r4
and r2, r5, r6
or r3, r7, r8
xor r4, r9,r10
sll r5,r11, r2
srl r6,r12, r0
sra r7, r1, r2
slt r8, r0, r2
addi r9, r3, -1
andi r10, r1, 15
ori r11,r13, 2
xori r12,r14, 15
slli r13,r15, 1
srli r14,r15, 1

波形3


clk
pc
instr
alu
mem
register
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
Now
650 ns
700 ns
750 ns
800 ns
850 ns
900 ns
950 ns
srai r15,r15, 1
slti r0, r1, 1
li r0, 4
li r1, 0
li r2, 0
load r3,0(r2)
addi r2, r2, 1
addi r0, r0, -1
add r1, r1, r3
bnz r0, loop
load r3,0(r2)
addi r2, r2, 1
addi r0, r0, -1
add r1, r1, r3
bnz r0, loop
load r3,0(r2)

波形 4


clk
pc
instr
alu
mem
register
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
Now
1000 ns
1050 ns
1100 ns
1150 ns
1200 ns
1250 ns

SpringCPU の拡張

1. 次の２つの命令を SpringCPU に実装せよ

```
23. call rd, subroutine # function invocation,
                        # save return address to rd
24. ret  rs1            # return from function,
                        # return address is in rs1
```

2. 16-bit SpringCPU16 を実装せよ

```
program counter:     16 bits
data:                16 bits
instruction memory: 2^16 = 64K = 65536
data memory:        2^16 = 64K = 65536
```