Aardvark - 8-bit Computer Processor Design

JEONG Andy, MACSHANE Gordon, SO Brenda

Contents

1 ABSTRACT

Employing a subset of MIPS ISA, an 8-bit processor was designed using Verilog language. Each instruction is set to 1-byte, or 8-bit, length, and the main memory is word addressable while the instruction memory is byte addressable. There are 4 addressable registers: \$s1 (00), \$s2 (01), \$sp (10) and \$ra (11). Two other registers in the register file, \$slt_0 and \$slt_1, cannot be addressed directly, but may be well utilized for jumping (beq instruction). All registers are initialized at 0, while the \$sp pointer is initialized at 0xFF (255 in decimal), which is the top of the memory stack.

2 MEMORY

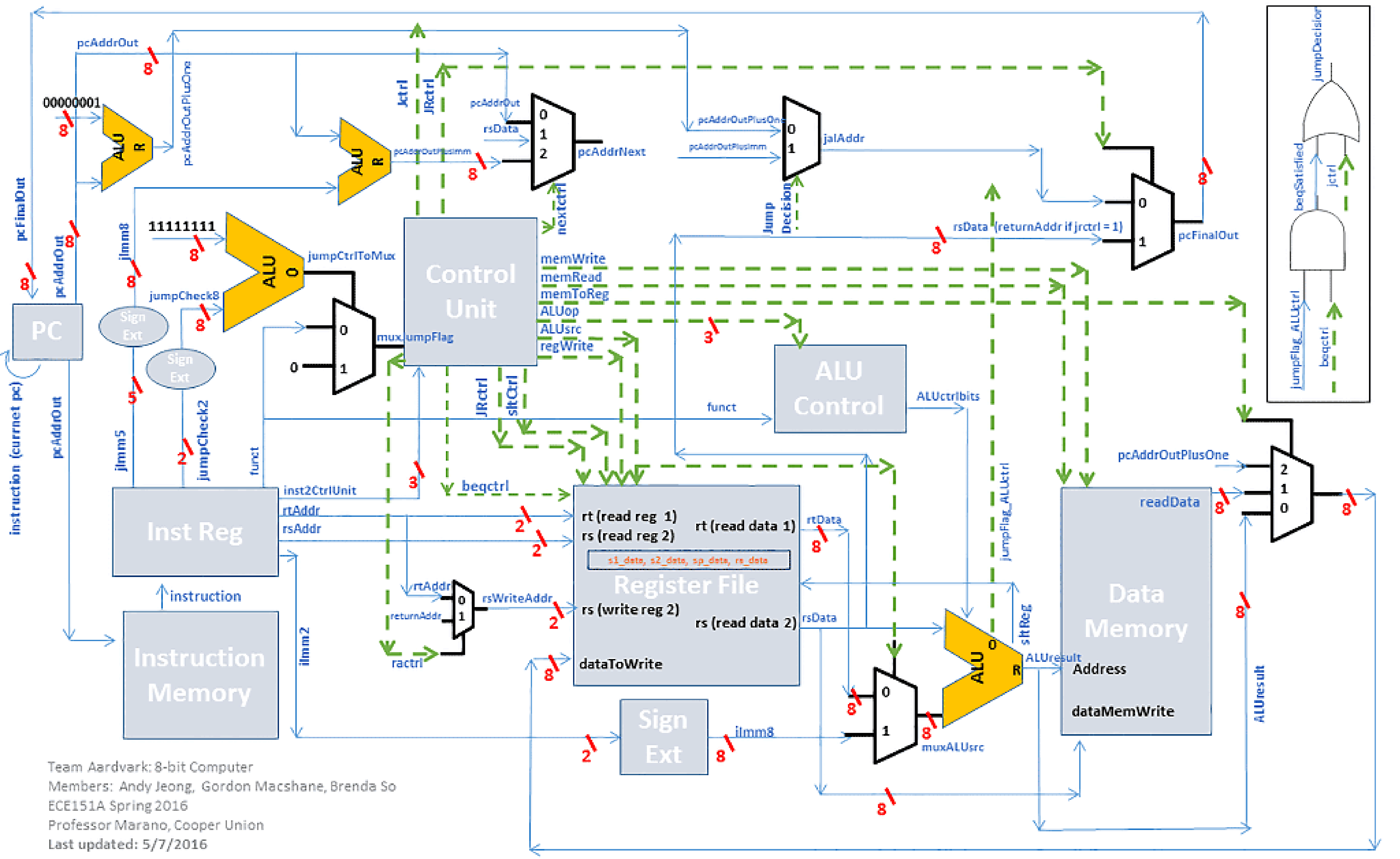
The memory is byte addressable, since all registers can only carry 8 bits and each instruction has 8 bits. There are three sections of the memory stack, the text (0x00 - 0x4B), the data (0x4C - 0x4F), and the stack (0x50 - 0xFF). The allocation of text section is determined to be 64 bytes of instruction because the programs presented in general involve at most 28 lines, and as the maximum 64 was enough. The data section is limited to only 4 global bytes in general no more than 4 arguments will be taken for most programs (unless very large programs, which are not considered for this processor).

Since the fact that a pseudo-instruction–load address(”la”)–needs to handle loading of data into stack, the assembler was written for this particular processor. If a programmer declared a global variable in the data section of the memory, the pseudo instruction would need to be expanded into the instructions available in the core instruction set. In the worst case scenario, each ”la” command would need to be replaced by 16 core instructions. Since the text section can only occupy at most 76 bytes of instructions, there could only be at most 4 bytes for global data, assuming each data is one byte.

3 ASSEMBLER

In order to deliver the assembly code to Verilog to compute, the assembly language had to be converted to binary machine code. To that end, an assembler with a linker were implemented in C language. The first portion of the assembler takes in the assembly file ”input.s” and outputs a link file, which is then processed through our second part, which then fully converts it to absolute memory-addressable binary. The output of this is then fed into the Verilog code.

4 PROCESSOR DIAGRAM



Team Aardvark: 8-bit Computer
Members: Andy Jeong, Gordon Macshane, Brenda So
ECE151A Spring 2016
Professor Marano, Cooper Union
Last updated: 5/7/2016

5 SAMPLE CODE

To demonstrate the functionality of the instruction set, three sets of codes were written. The first piece of code demonstrates basic arithmetic operations, including addition, subtraction, multiplication and division. The second and third sets of code demonstrates the summation from 1 to 10. The former performs the procedure recursively, the second performs the procedure non-recursively. Complementary C code is also provided for comparison.

5.1 ARITHMETIC CODE

5.1.1 Arithmetic Code in C

```
int mult(int a, int b)
  {
      int c = a * b;
      return c;
  }
  int div(int a, int b)
  {
      int c = a / b;
      return c;
  }
```

5.1.2 Arithmetic Code in Assembly

```
.data
.byte num1 2
.byte num2 3
.text
la $sp num1
lw 0 $s1
lw 1 $s2
nand $sp $ra
sw 0 $s1
lw -1 $s1
addi -1 $sp
jal MULT
ADDITION: lw 1 $s2
add $s1 $s2
lw 0 $s2
MULT: addi -1 $s2
sw 0 $s2
sw -1 $ra
slt_0 $ra $s2
beq ADDITION
addi -1 $sp
sw 0 $s1
___________________________________________
#a = $s1, b = $s2
```

```
sw 0, $ra
sw 1, $ra
sw 2, $s2
sw 3, $s3

subtraction:
lw 3, $s1
nand $s2, $s2
addi 1, $s2
add $s1, $s2
sw 3, $s1

division:
lw 3, $s2 #check if it has subtracted enough times
slt_0 $ra, $s2 # 0 < sum
lw 1, $s1
addi 1, $s1
sw 1, $s1
lw 2, $s2 #put b into s2 for subtraction
beq subtraction
lw 1, $s1
addi -1, $s1 #it always overshoots by one
sw 1, $s1
lw 1, $ra
```

5.2 NON-RECURSIVE CODE

5.2.1 Non-Recursive Code in C

```
function(int n)
  {
      int i, sum;
      int n = 10;
      for (i = 0; i < n; i++)
      {
          sum += i;
      }
  }
```

5.2.2 Non-Recursive Code in Assembly

```
function:

nand $s1, $s1
nand $s2, $s2
sl $s1, $s1
nand $s2 $s1     # i($s2) = 1
sw 0, $s2        # store i
```

```
nand $s1, $s1    #s1 becomes 1
sr $s1, $s1      #initialize sum($s1) = 0
sw 1, $s1        #store sum($s1) into offset 1 of $sp

add $s1, $s2      #n = 1
sl $s1, $s1
sl $s1, $s1
sl $s1, $s1       #n = 2^3 * n
add $s1, $s2      #n = 9
add $s1, $s2      #n($s1) = 10
sw 2, $s1        #save 10 to memory

nand $s1, $s1
nand $s2, $s2
sl $s1, $s1
nand $s1 $s2              # $s1 = 1
sw 3, $s1                 #store value 1

LOOP:
lw 2, $s1         #load n
slt_0 $s2, $s1    # i < n
slt_1 $s1, $s2    # i > n
beq END          #if i = n, go to END
lw 0, $s2        #load i
lw 1, $s1        #load sum
add $s1, $s2     #sum($s1) += i
sw 1, $s1
lw 3, $s1
lw 0, $s2        #load i to $s2
add $s2, $s1     # i = i + 1
jal LOOP
END:
lw 1, $s1
```

5.3 RECURSIVE CODE

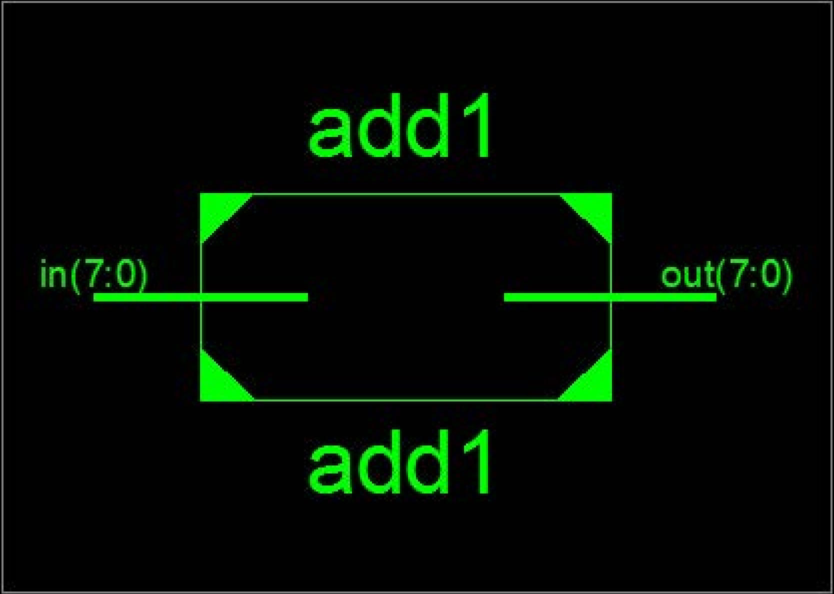
5.3.1 Recursive Code in C

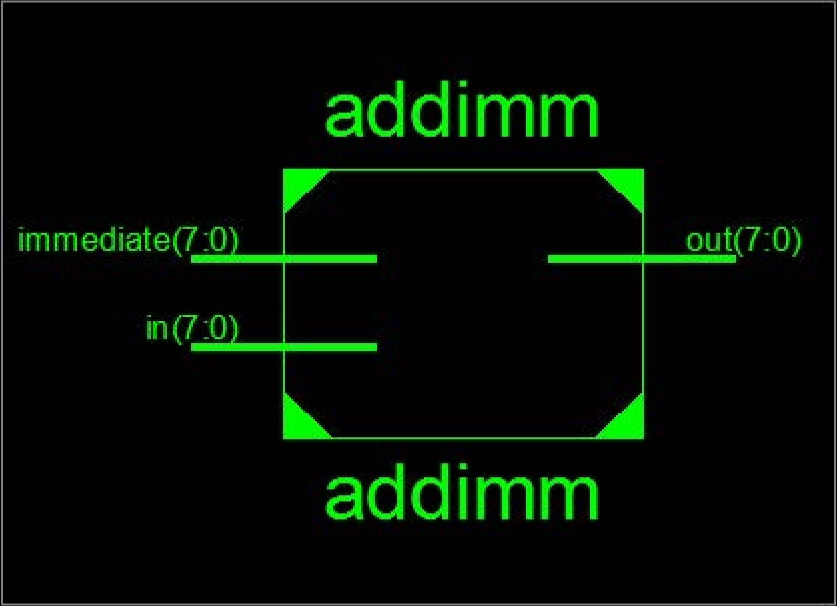
```
int sum(int n)
  {
      if (n > 1){
          return n+sum(n-1);
      }
      return 1;
  }
```

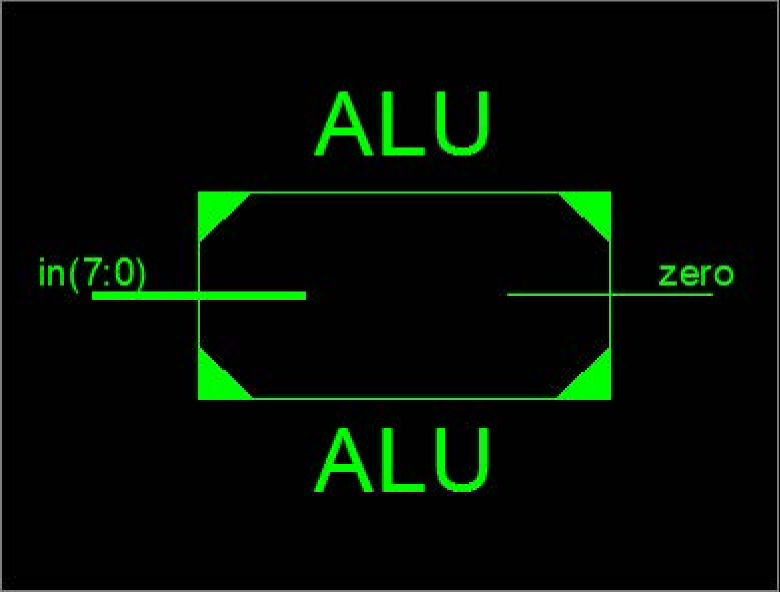
5.3.2 Recursive Code in Assembly

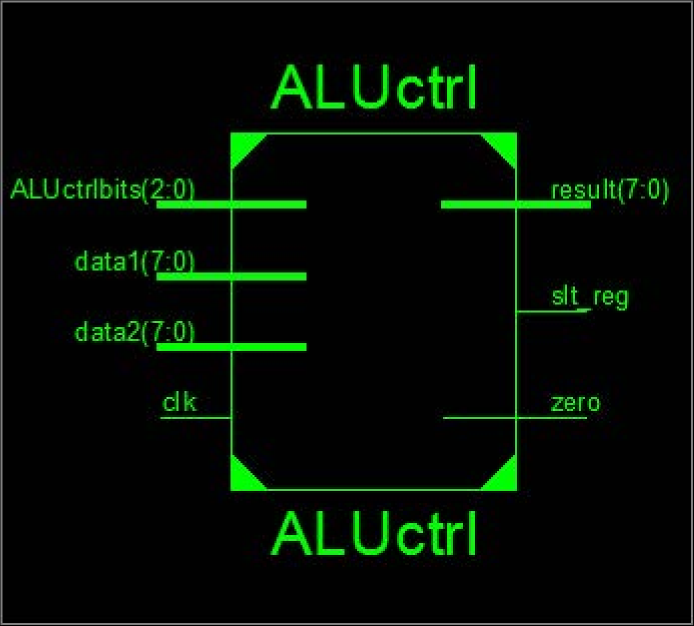
```
function:
nand $s1, $s1
nand $s2, $s2
sl $s1, $s1
nand $s2 $s1
nand $s1, $s1              #$s1 = -2
sw 0, $s2                  #save n in $s2 to memory

#then follows to sum (recursion)
sum:
add $sp, $s1               #adjust stack for 2 items $s1 = -2
sw 1, $ra                  #save return address to memory
sw 2, $s2                  #save current n to memory
nand $s1, $s1              #s1 becomes 1
sw 3, $s1                  #saves $s1 (= 1)
slt_0 $s1, $s2             #slt_0=n > 1?, 0:1
slt_1 $s2, $s1             #slt_1=n>1?, 0:1
beq END                    #if n < 1, jump to L1

nand $s1, $s1              #$s1 = -2
lw  3, $s2                 #load 1 into $s2
add $s1, $s2               #$s1 = -1
lw 2, $s2                  #load n into $s2
add $s2, $s1               #decrement n by 1
jal sum                    #recursive call

lw 3, $s1
add $sp, $s1               #pop 2 items from stack (s1 needs to be 1)
add $sp, $s1
lw 1, $ra                  #restore return address
lw 0, $s1                  #restore original n to s1
add $s2, $s1               #s2 = s2 + s1
jr $ra                     #return to calling address

END:
jr $ra                     #return 0 if n is not > 1
```

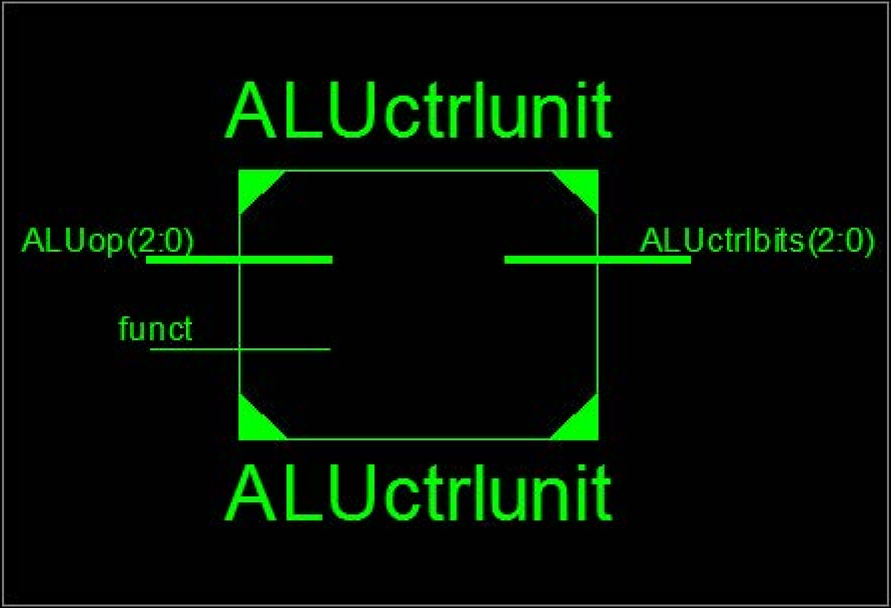
6 MODULE I/O

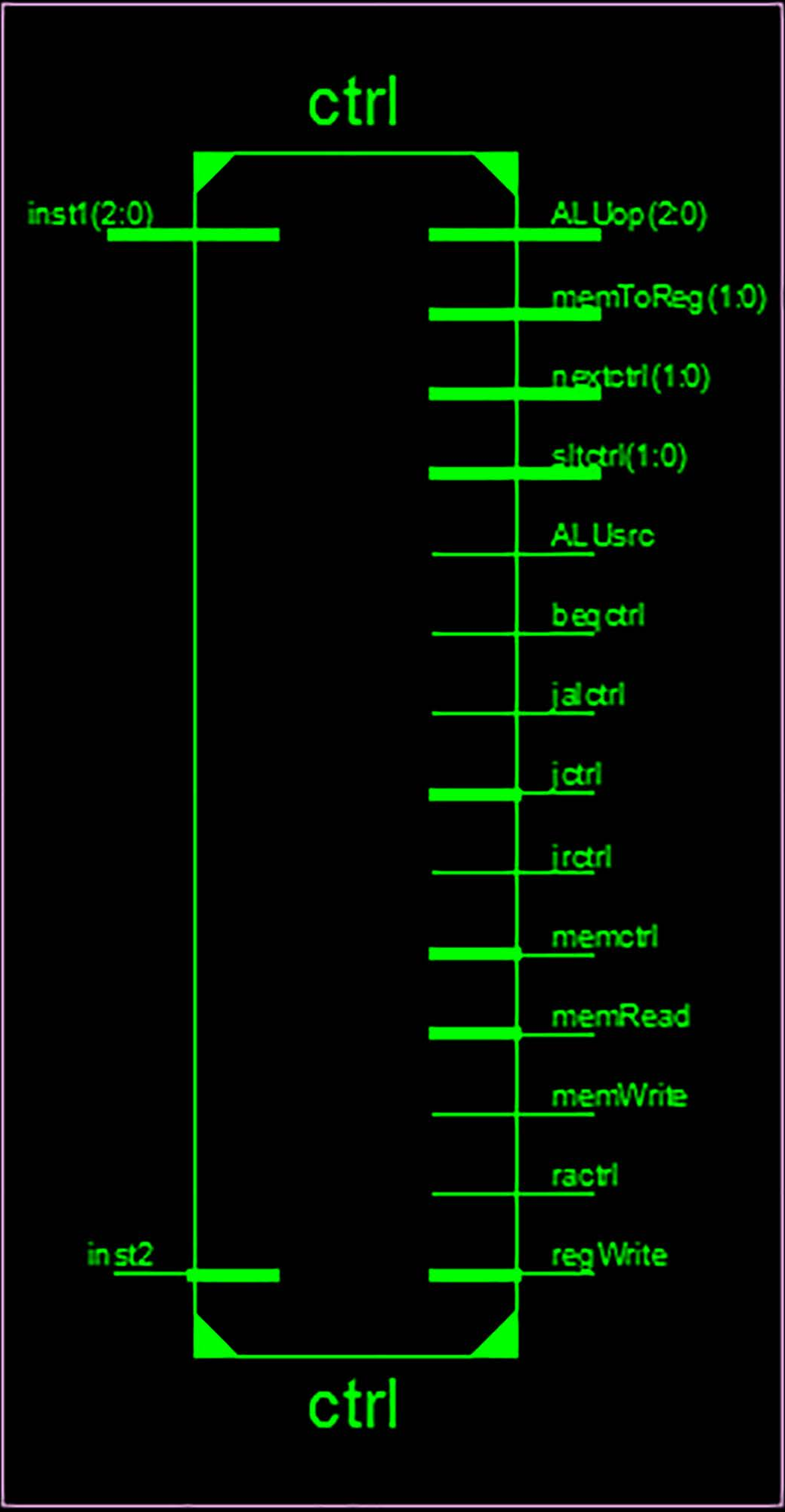

add1
in(7:0)
out(7:0)
add1

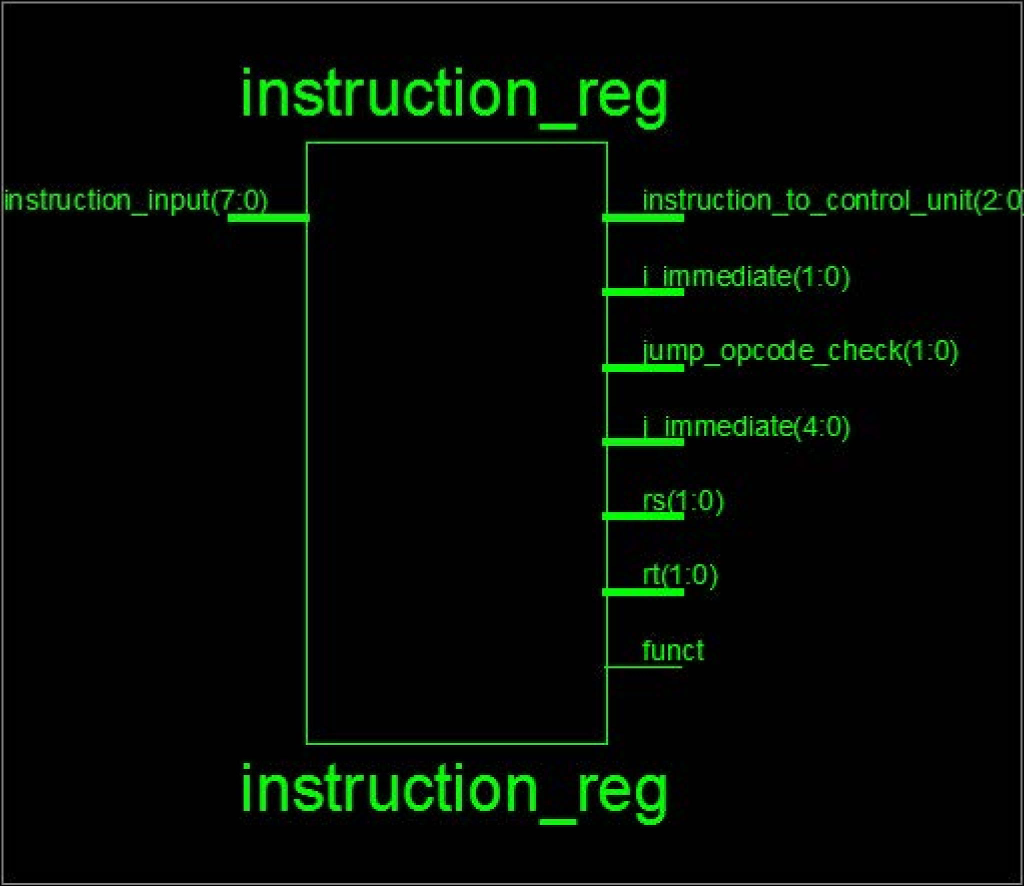

addimm
immediate(7:0)
in(7:0)
out(7:0)
addimm

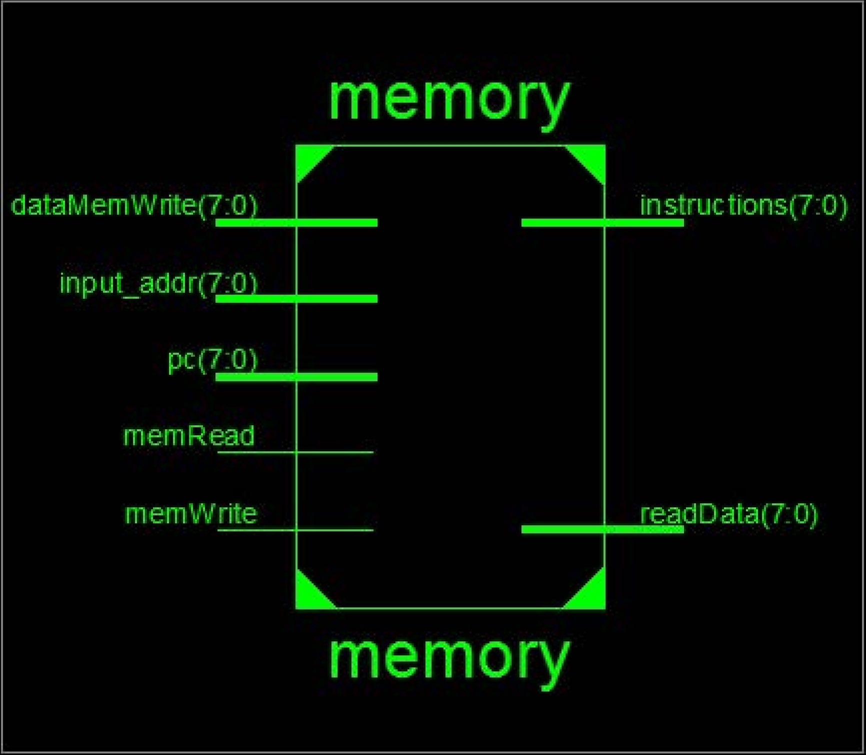

ALU
in(7:0)
zero
ALU

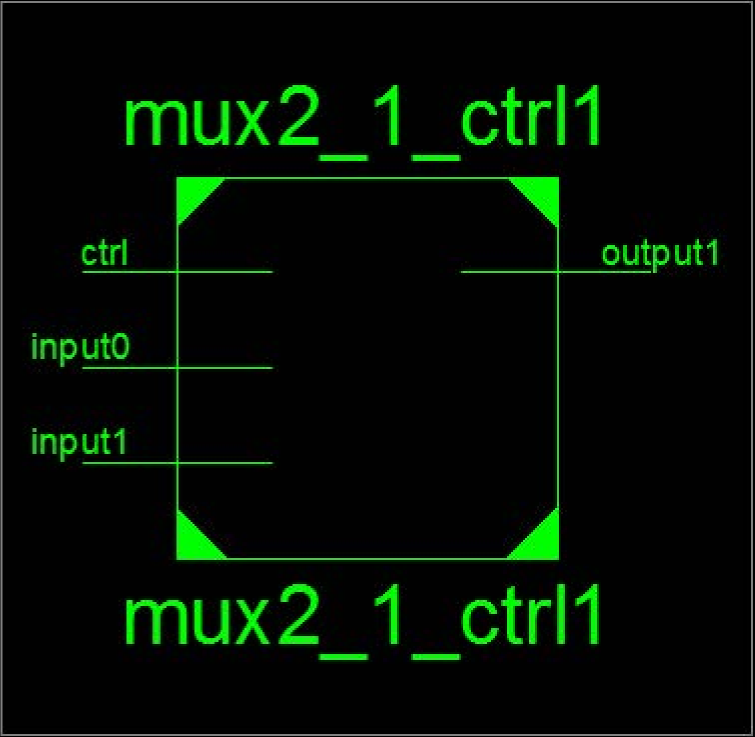

ALUctrl
ALUctrlbits(2:0)
data1(7:0)
data2(7:0)
clk
result(7:0)
slt_reg
zero
ALUctrl

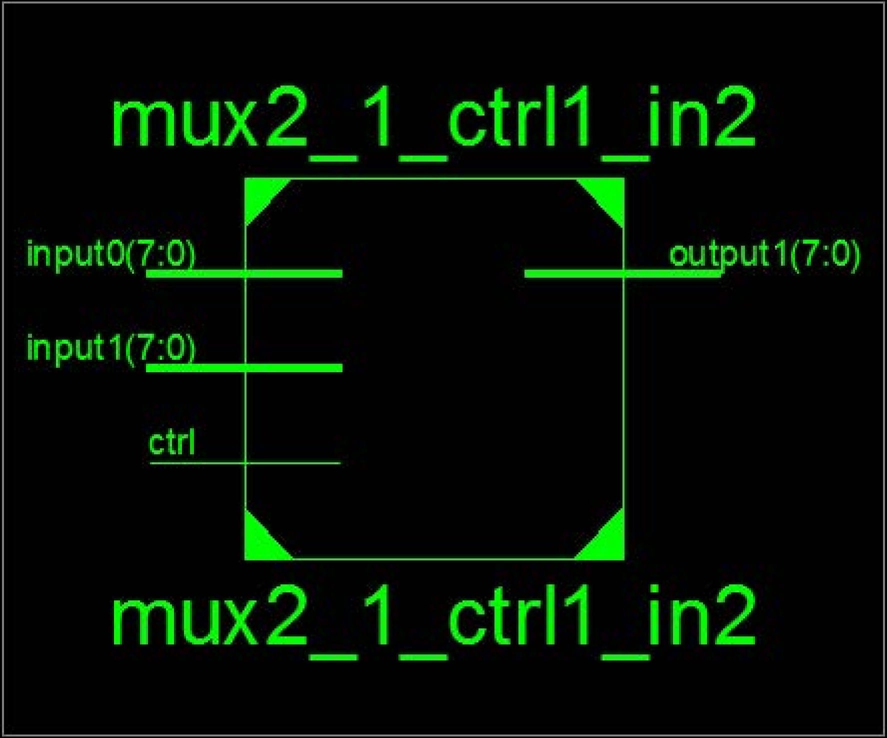

ALUctrlunit
ALUop(2:0)
funct
ALUctrlbits(2:0)
ALUctrlunit

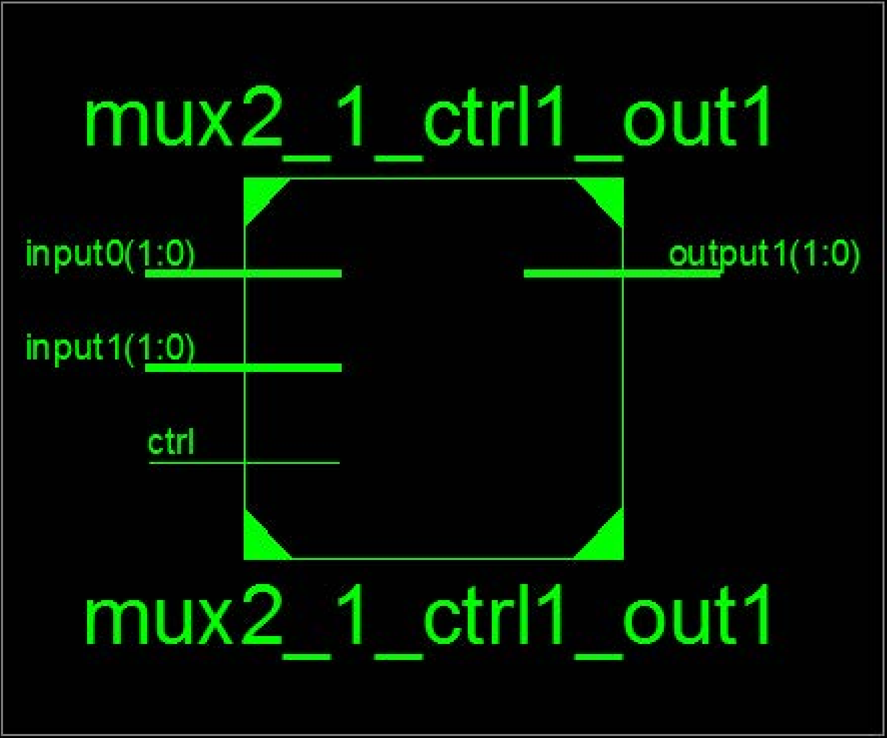

ctrl
inst1(2:0)
inst2
ALUop(2:0)
memToReg(1:0)
nextctrl(1:0)
sltctrl(1:0)
ALUsrc
beqctrl
jalctrl
jctrl
jrctrl
memctrl
memRead
memWrite
ractrl
regWrite
ctrl

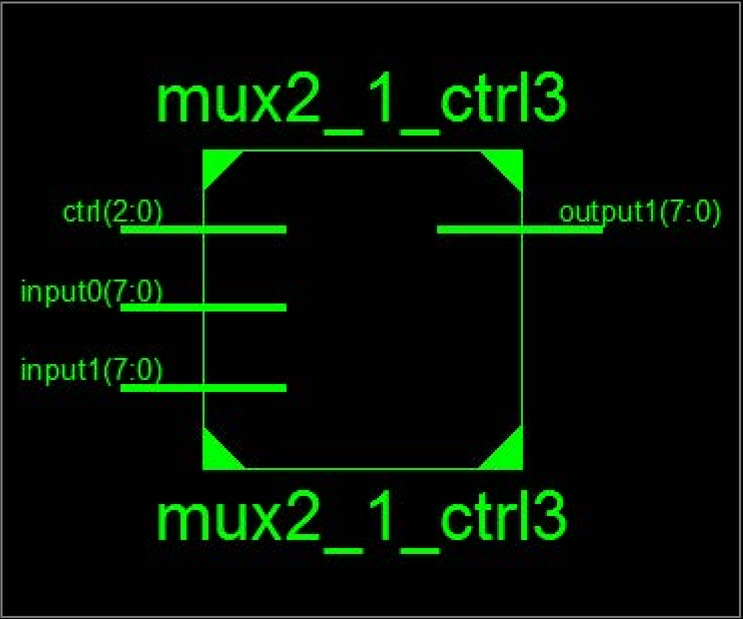

instruction_reg
instruction_input(7:0)
instruction_to_control_unit(2:0)
i_immediate(1:0)
jump_opcode_check(1:0)
j_immediate(4:0)
rs(1:0)
rt(1:0)
funct
instruction_reg

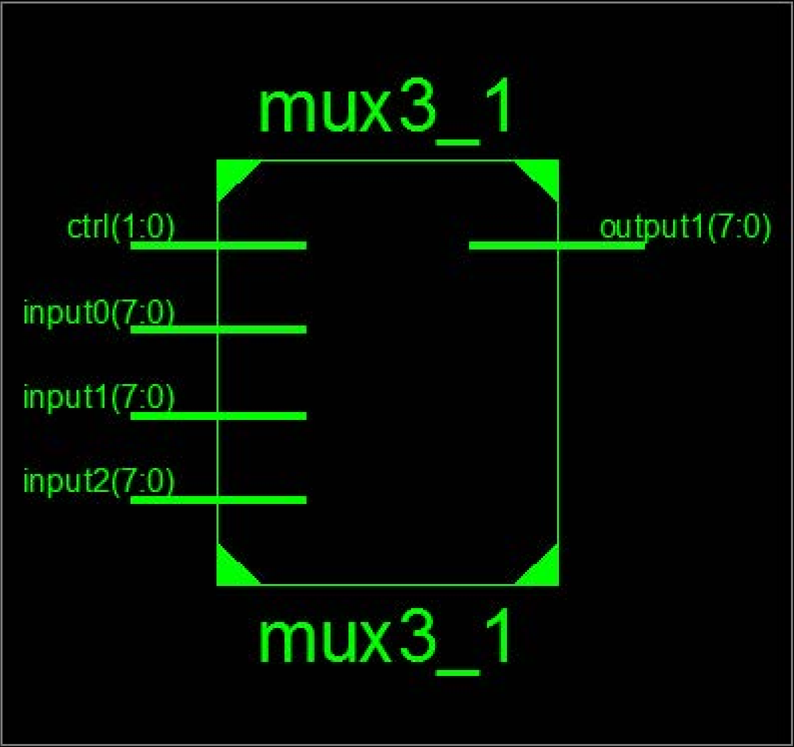

memory
dataMemWrite(7:0)
input_addr(7:0)
pc(7:0)
memRead
memWrite
instructions(7:0)
readData(7:0)
memory

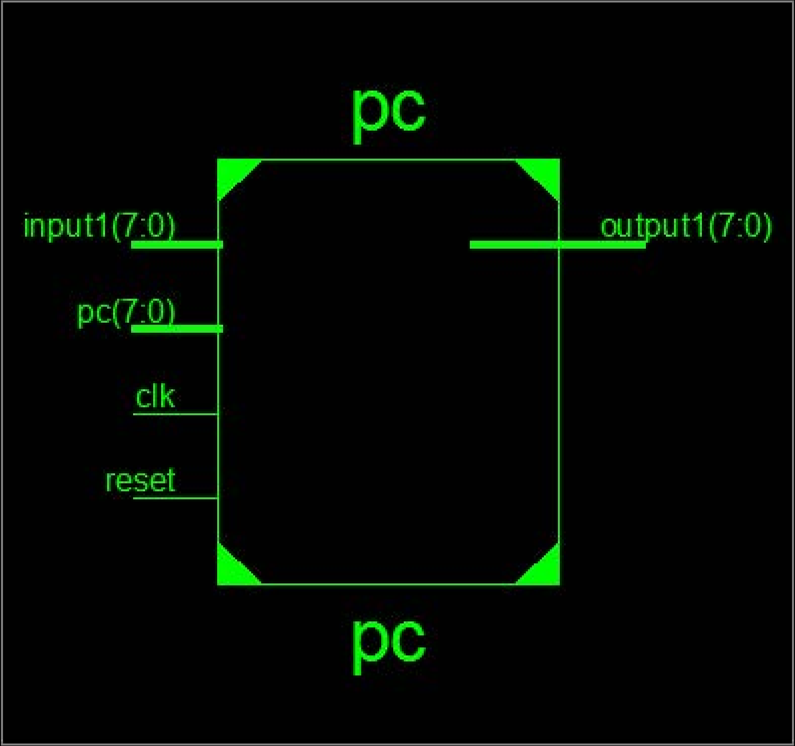

mux2_1_ctrl1
ctrl
input0
input1
output1
mux2_1_ctrl1

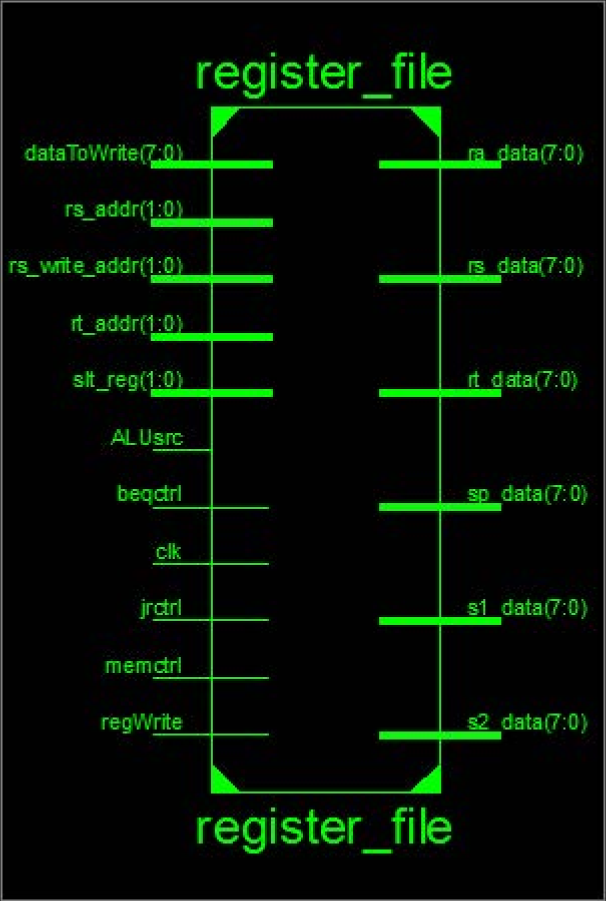

mux2_1_ctrl1_in2
input0(7:0)
input1(7:0)
ctrl
output1(7:0)
mux2_1_ctrl1_in2

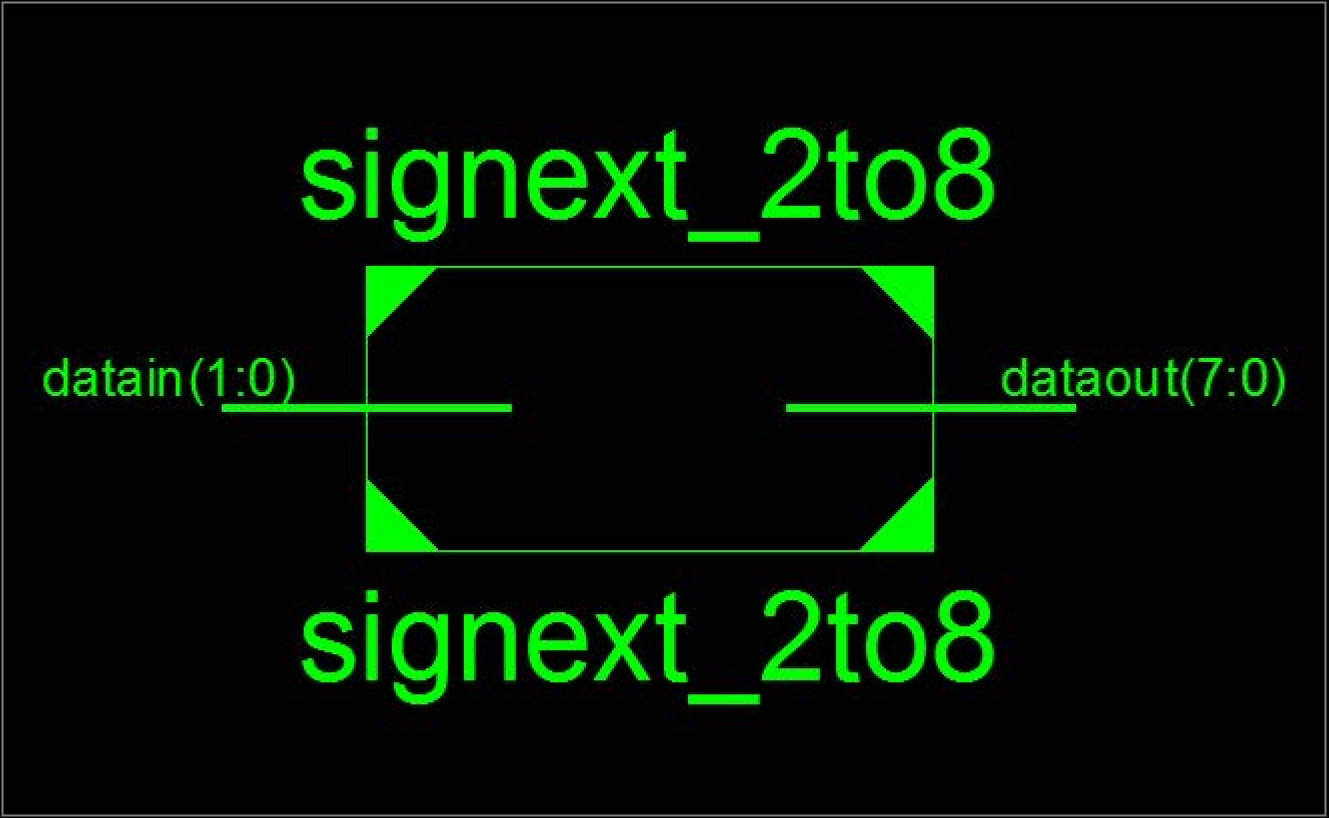

mux2_1_ctrl1_out1
input0(1:0)
input1(1:0)
ctrl
output1(1:0)
mux2_1_ctrl1_out1

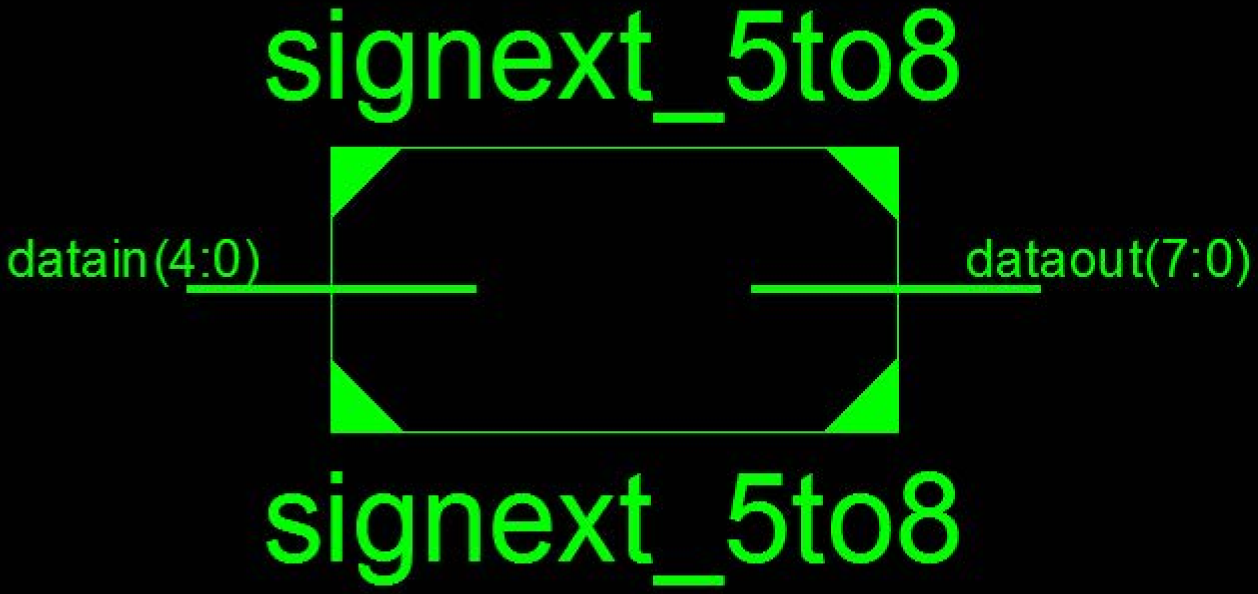

mux2_1_ctrl3
ctrl(2:0)
input0(7:0)
input1(7:0)
output1(7:0)
mux2_1_ctrl3


mux3_1
ctrl(1:0)
input0(7:0)
input1(7:0)
input2(7:0)
output1(7:0)
mux3_1


pc
input1(7:0)
pc(7:0)
clk
reset
output1(7:0)
pc


register_file
dataToWrite(7:0)
rs_addr(1:0)
rs_write_addr(1:0)
rt_addr(1:0)
slt_reg(1:0)
ALUsrc
beqctrl
clk
jrctrl
memctrl
regWrite
ra_data(7:0)
rs_data(7:0)
rt_data(7:0)
sp_data(7:0)
s1_data(7:0)
s2_data(7:0)
register_file


signext_2to8
datain(1:0)
dataout(7:0)
signext_2to8


signext_5to8
datain(4:0)
dataout(7:0)
signext_5to8