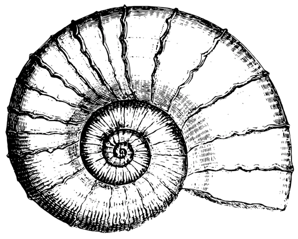
The Shell CPU

Datapath Verification



Jonathan Rice Shelley II

November 4, 2020

Version 1.0

Contents

1 Datapath Verification Overview 2

2 General Instruction Testing 3

2.1 Add type RRR instruction verification . . . 3
2.2 Sub type RRR instruction verification . . . 4
2.3 And instruction verification . . . 5
2.4 Or instruction verification . . . 6
2.5 Xor instruction verification . . . 7
2.6 Nand instruction verification . . . 8
2.7 Asr instruction verification . . . 9
2.8 Asl instruction verification . . . 10
2.9 Addi instruction verification . . . 11
2.10 Subi instruction verification . . . 12
2.11 Lui instruction verification . . . 13

3 Branching Instruction Testing 14

3.1 CMP instruction verification . . . 14
3.2 Beq instruction verification . . . 15
3.3 Jalr instruction verification . . . 16

4 CPU Test Program 17

1 Datapath Verification Overview

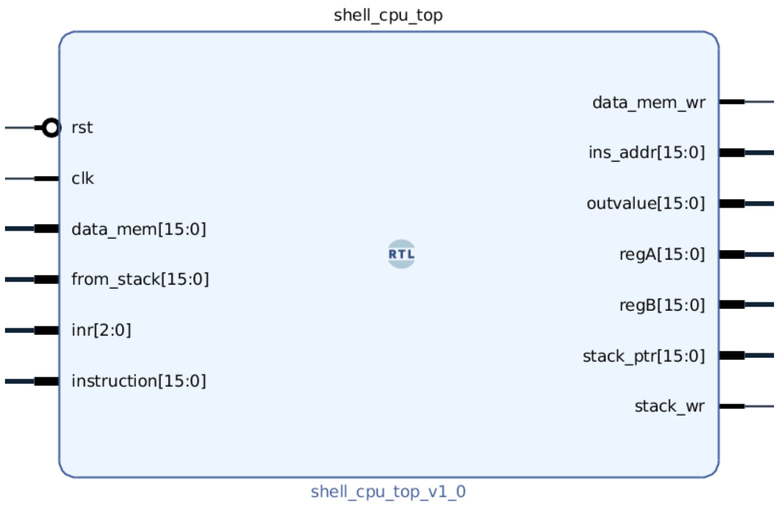


Figure 1: Top level CPU design

This document details the datapath verification of the Shell CPU. All the datapath components are connected in a top-level component module. The top level Shell CPU module is shown above in Figure 1. The VHDL for this module is to long to be listed and can be found with other documentation here. The top level CPU design can be easily implemented into a larger project like the one shown in Figure 2 to add soft core functionality to a design.

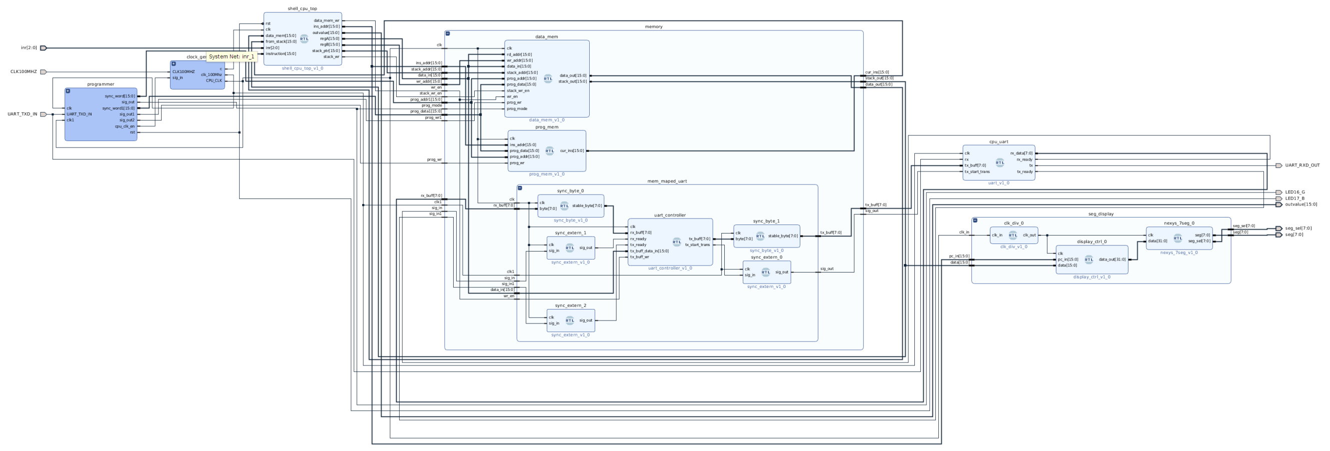


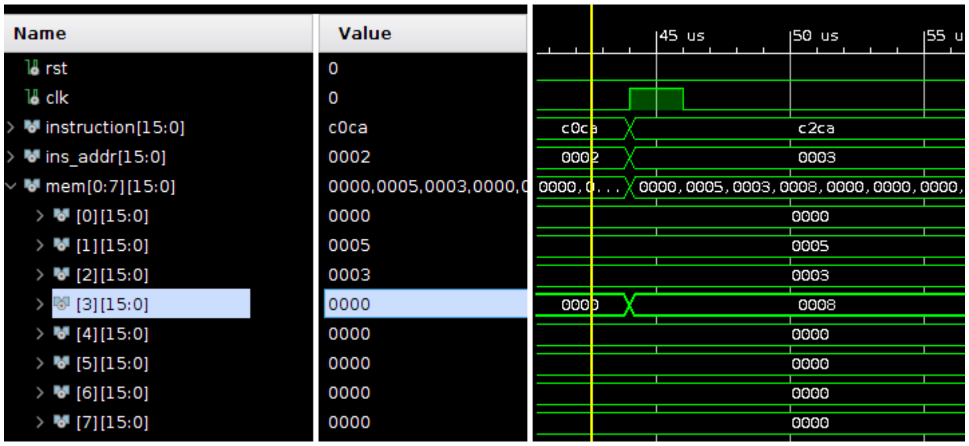
Figure 2: Example CPU implementation with memory mapped UART hardware / UART programmer

2 General Instruction Testing

A test bench is created to test ”general” Shell CPU instructions. General in this case refers to instructions that can be easily tested by viewing the contents of the CPUs register file. Instructions that require a more intensive testing strategy will be verified in a separate section.

2.1 Add type RRR instruction verification

Instruction	Machine Code
add r3, r1, r2	1100000011001010



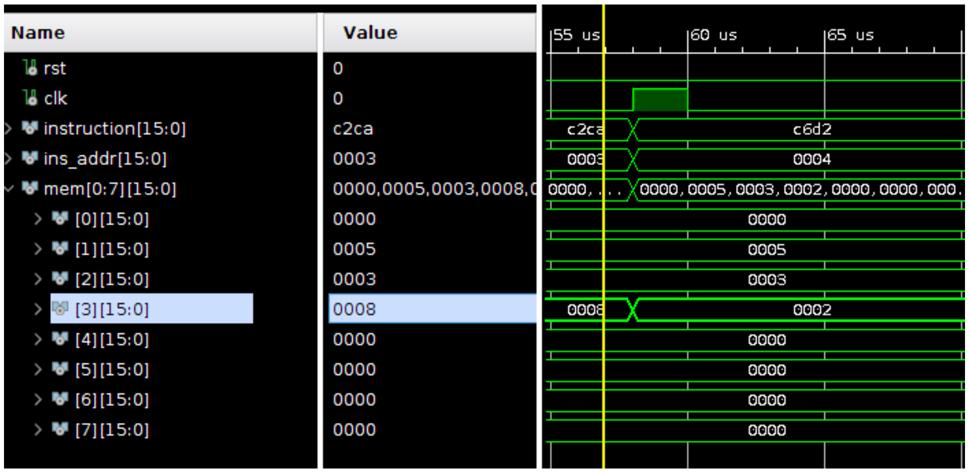
Name	Value
rst	0
clk	0
instruction[15:0]	c0ca
ins_addr[15:0]	0002
mem[0:7][15:0]	0000,0005,0003,0000,0
[0][15:0]	0000
[1][15:0]	0005
[2][15:0]	0003
[3][15:0]	0000
[4][15:0]	0000
[5][15:0]	0000
[6][15:0]	0000
[7][15:0]	0000

Figure 3: Add RRR instruction simulation

A simulation of the Add RRR instruction is shown above in Figure 3. The instruction performs the addition of register one and register two. The sum is stored in register three. In the simulation shown above r1 and r2 can be seen to be five and three respectively. After the positive clock edge the add instruction is executed and eight is seen written to register three. The functionality is verified because the sum of five and three is eight.

2.2 Sub type RRR instruction verification

Instruction	Machine Code
sub r3, r1, r2	1100001011001010



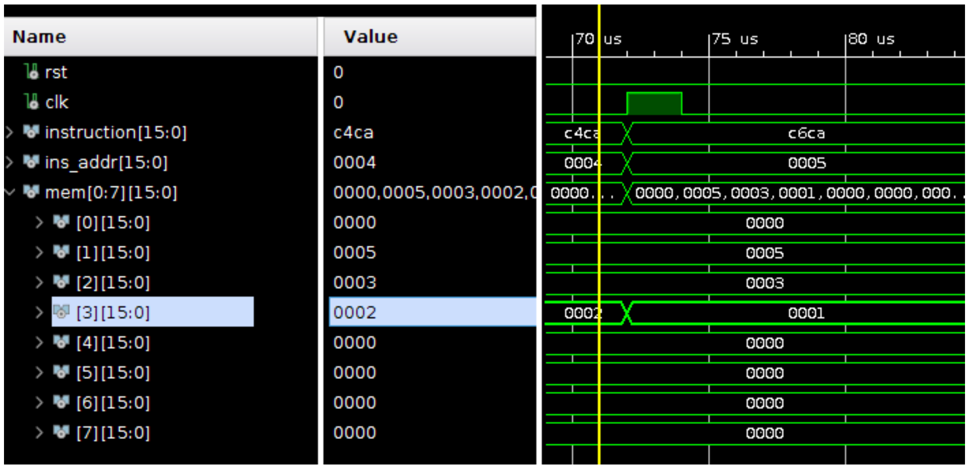
Name	Value
rst	0
clk	0
instruction[15:0]	c2ca
ins_addr[15:0]	0003
mem[0:7][15:0]	0000,0005,0003,0008,0
[0][15:0]	0000
[1][15:0]	0005
[2][15:0]	0003
[3][15:0]	0008
[4][15:0]	0000
[5][15:0]	0000
[6][15:0]	0000
[7][15:0]	0000

Figure 4: Sub RRR instruction simulation

A simulation of the Sub RRR instruction is shown above in Figure 4. The instruction performs the subtraction of register one and register two. The difference is stored in register three. In the simulation shown above r1 and r2 can be seen to contain five and three respectively. After the positive clock edge the sub instruction is executed and two is seen written to register three. The functionality is verified because the difference of five and three is two.

2.3 And instruction verification

Instruction	Machine Code
and r3, r1, r2	1100010011001010



Name	Value
rst	0
clk	0
instruction[15:0]	c4ca
ins_addr[15:0]	0004
mem[0:7][15:0]	0000,0005,0003,0002,0
[0][15:0]	0000
[1][15:0]	0005
[2][15:0]	0003
[3][15:0]	0002
[4][15:0]	0000
[5][15:0]	0000
[6][15:0]	0000
[7][15:0]	0000

Figure 5: And instruction simulation

A simulation of the and instruction is shown above in Figure 5. The instruction performs the bit wise and of register one and register two. The result is stored in register three. In the simulation shown above r1 and r2 can be seen to contain five and three respectively. After the positive clock edge the and instruction is executed and one is seen written to register three. The functionality is verified because the bit wise and of five and three is one.

2.4 Or instruction verification

Instruction	Machine Code
or r3, r1, r2	1100011011001010

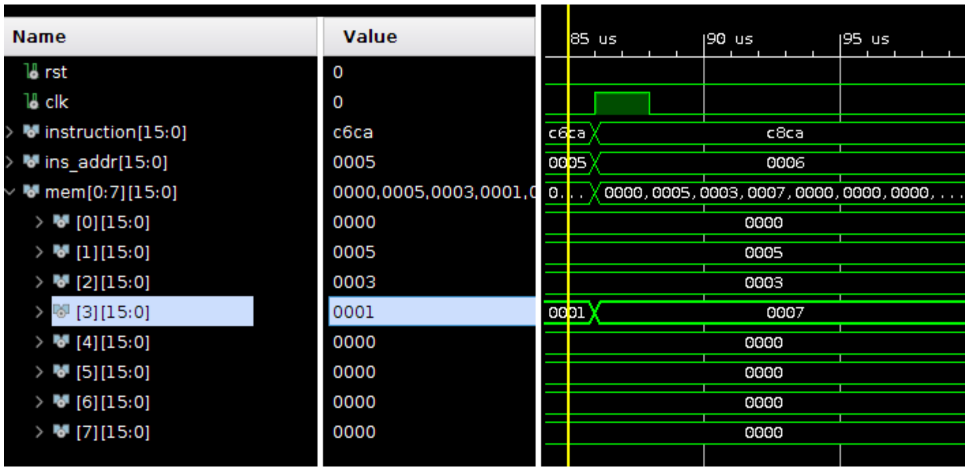
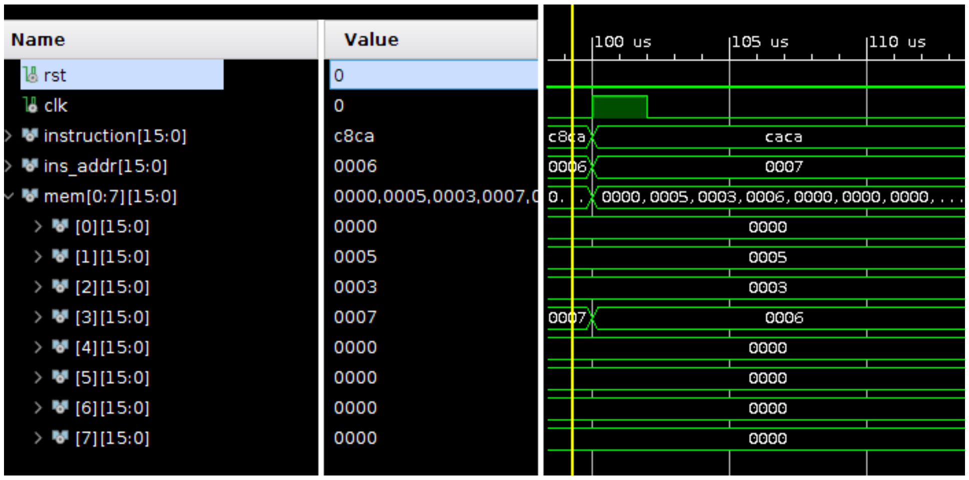


Figure 6: Or instruction simulation

A simulation of the or instruction is shown above in Figure 6. The instruction performs the bit wise or of register one and register two. The result is stored in register three. In the simulation shown above r1 and r2 can be seen to contain five and three respectively. After the positive clock edge the or instruction is executed and seven is seen written to register three. The functionality is verified because the bit wise or of five and three is seven.

2.5 Xor instruction verification

Instruction	Machine Code
xor r3, r1, r2	1100100011001010



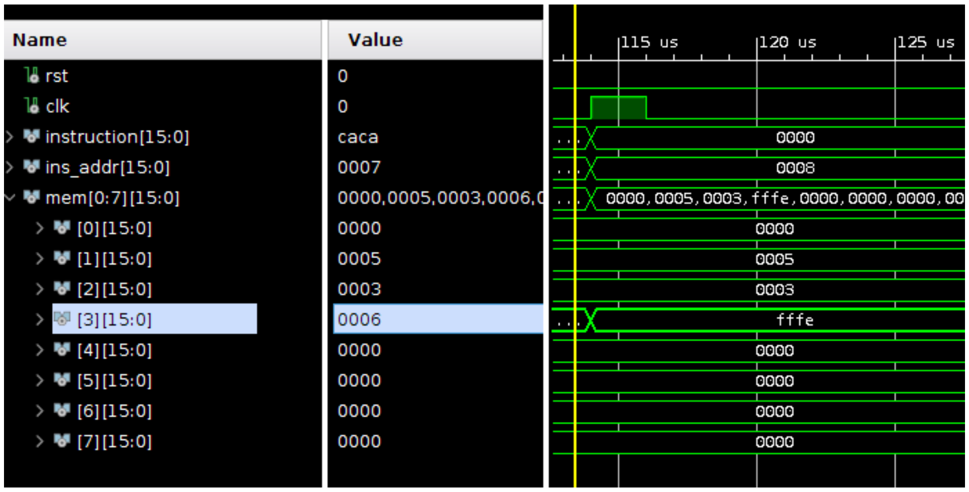
Name	Value
rst	0
clk	0
instruction[15:0]	c8ca
ins_addr[15:0]	0006
mem[0:7][15:0]	0000,0005,0003,0007,0
[0][15:0]	0000
[1][15:0]	0005
[2][15:0]	0003
[3][15:0]	0007
[4][15:0]	0000
[5][15:0]	0000
[6][15:0]	0000
[7][15:0]	0000

Figure 7: Xor instruction simulation

A simulation of the xor instruction is shown above in Figure 7. The instruction performs the bit wise xor of register one and register two. The result is stored in register three. In the simulation shown above r1 and r2 can be seen to contain five and three respectively. After the positive clock edge the xor instruction is executed and six is seen written to register three. The functionality is verified because the bit wise xor of five and three is six.

2.6 Nand instruction verification

Instruction	Machine Code
nand r3, r1, r2	1100101011001010



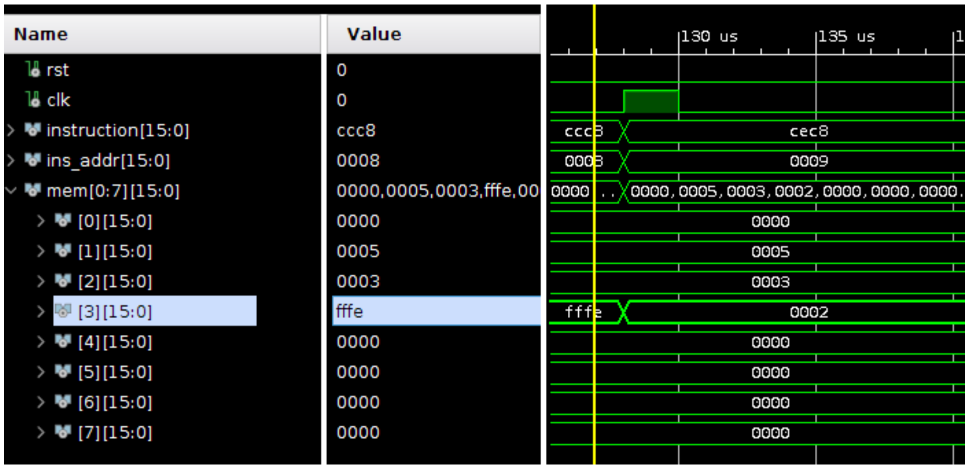
Name	Value
rst	0
clk	0
instruction[15:0]	caca
ins_addr[15:0]	0007
mem[0:7][15:0]	0000,0005,0003,0006,0
[0][15:0]	0000
[1][15:0]	0005
[2][15:0]	0003
[3][15:0]	0006
[4][15:0]	0000
[5][15:0]	0000
[6][15:0]	0000
[7][15:0]	0000

Figure 8: Nand instruction simulation

A simulation of the nand instruction is shown above in Figure 8. The instruction performs the bit wise nand of register one and register two. The result is stored in register three. In the simulation shown above r1 and r2 can be seen to contain five and three respectively. After the positive clock edge the nand instruction is executed and 0xfffe is seen written to register three. The functionality is verified because the bit wise nand of five and three is 0xfffe (16-bit hex).

2.7 Asr instruction verification

Instruction	Machine Code
asr r3, r1	1100110011001000



Name	Value
rst	0
clk	0
instruction[15:0]	ccc8
ins_addr[15:0]	0008
mem[0:7][15:0]	0000,0005,0003,fffe,00
[0][15:0]	0000
[1][15:0]	0005
[2][15:0]	0003
[3][15:0]	fffe
[4][15:0]	0000
[5][15:0]	0000
[6][15:0]	0000
[7][15:0]	0000

Figure 9: Asr instruction simulation

A simulation of the asr instruction is shown above in Figure 9. The instruction performs an arithmetic shift right on register one. The result is stored in register three. In the simulation shown above r1 contains a value of five. After the positive clock edge the asr instruction is executed and two is written to register three. The functionality is verified because the one bit arithmetic shift right of five is two.

2.8 Asl instruction verification

Instruction	Machine Code
asl r3, r1	1100111011001000

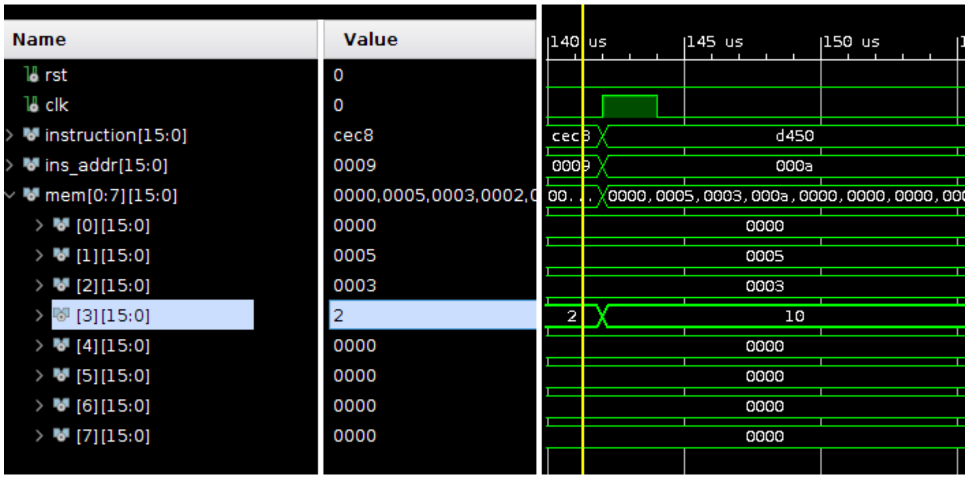
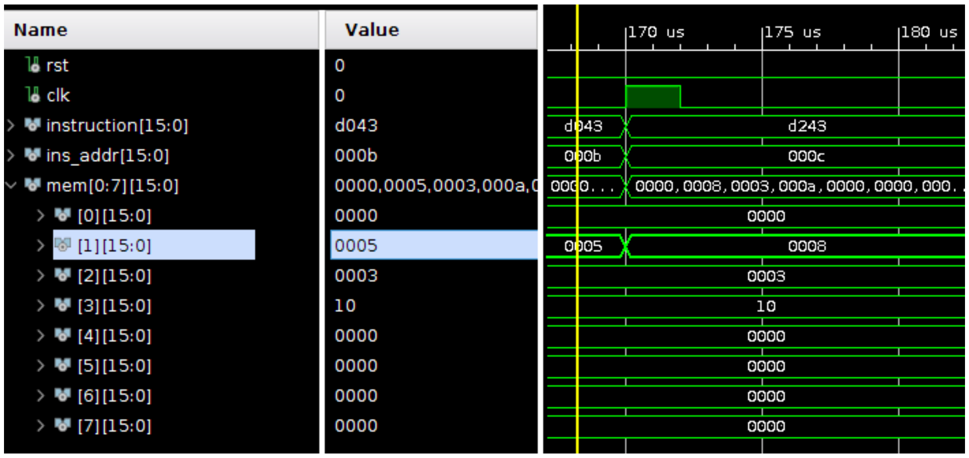


Figure 10: Asl instruction simulation

A simulation of the asl instruction is shown above in Figure 10. The instruction performs an arithmetic shift left on register one. The result is stored in register three. In the simulation shown above r1 contains a value of five. After the positive clock edge the asl instruction is executed and ten is written to register three. The functionality is verified because the one bit arithmetic shift left of five is ten.

2.9 Addi instruction verification

Instruction	Machine Code
addi r1, 3	1101000001000011



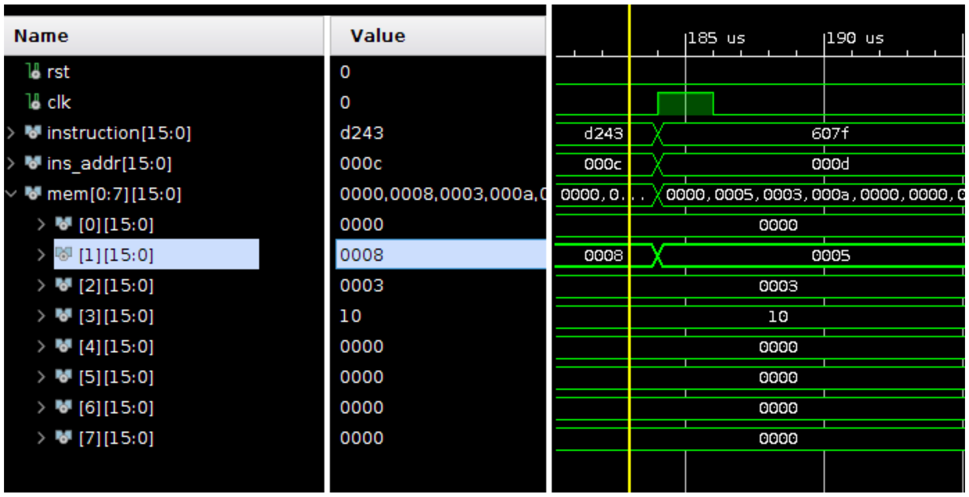
```
Name                Value
rst                 0
clk                 0
instruction[15:0]   d043
ins_addr[15:0]      000b
mem[0:7][15:0]      0000,0005,0003,000a,0
  [0][15:0]         0000
  [1][15:0]         0005
  [2][15:0]         0003
  [3][15:0]         10
  [4][15:0]         0000
  [5][15:0]         0000
  [6][15:0]         0000
  [7][15:0]         0000
```

Figure 11: Addi instruction simulation

A simulation of the addi instruction is shown above in Figure 11. The instruction adds the specified immediate three to the current value in r1. In the simulation, before the positive clock edge, r1 contains the value five after the clock edge r1 contains eight. Since five plus three is eight the instruction is working correctly.

2.10 Subi instruction verification

Instruction	Machine Code
subi r1, 3	1101001001000011



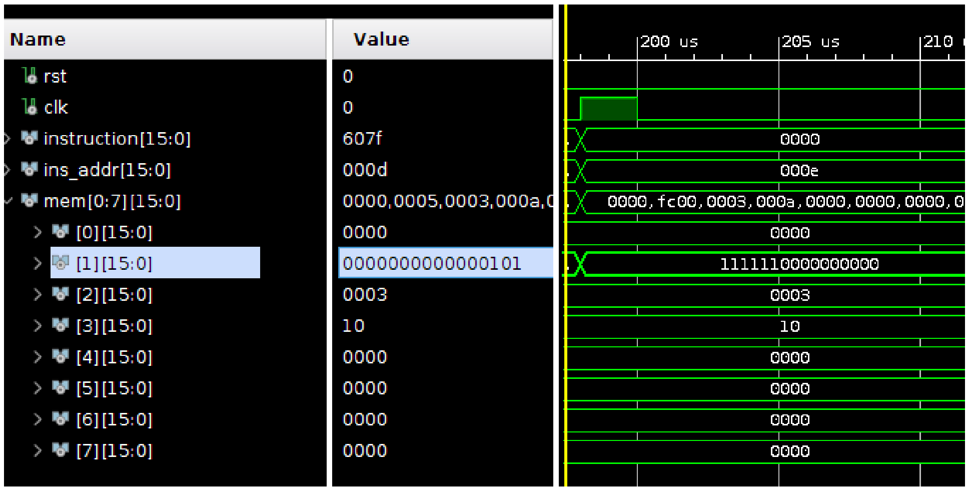
```
Name                 Value
rst                  0
clk                  0
instruction[15:0]    d243                       d243  607f
ins_addr[15:0]       000c                       000c  000d
mem[0:7][15:0]       0000,0008,0003,000a,0      0000,0...  0000,0005,0003,000a,0000,0000,0
  [0][15:0]          0000                       0000
  [1][15:0]          0008                       0008  0005
  [2][15:0]          0003                       0003
  [3][15:0]          10                         10
  [4][15:0]          0000                       0000
  [5][15:0]          0000                       0000
  [6][15:0]          0000                       0000
  [7][15:0]          0000                       0000
```

Figure 12: Subi instruction simulation

A simulation of the subi instruction is shown above in Figure 12. The instruction subtracts the specified immediate three from the current value in r1. In the simulation, before the positive clock edge, r1 contains the value eight after the clock edge r1 contains five. Since eight minus three is five the instruction is working correctly.

2.11 Lui instruction verification

Instruction	Machine Code
lui r1, 63	0110000001111111



Name	Value	200 us – 210 us
rst	0	
clk	0	
instruction[15:0]	607f	0000
ins_addr[15:0]	000d	000e
mem[0:7][15:0]	0000,0005,0003,000a,0	0000,fc00,0003,000a,0000,0000,0000,0
[0][15:0]	0000	0000
[1][15:0]	0000000000000101	1111110000000000
[2][15:0]	0003	0003
[3][15:0]	10	10
[4][15:0]	0000	0000
[5][15:0]	0000	0000
[6][15:0]	0000	0000
[7][15:0]	0000	0000

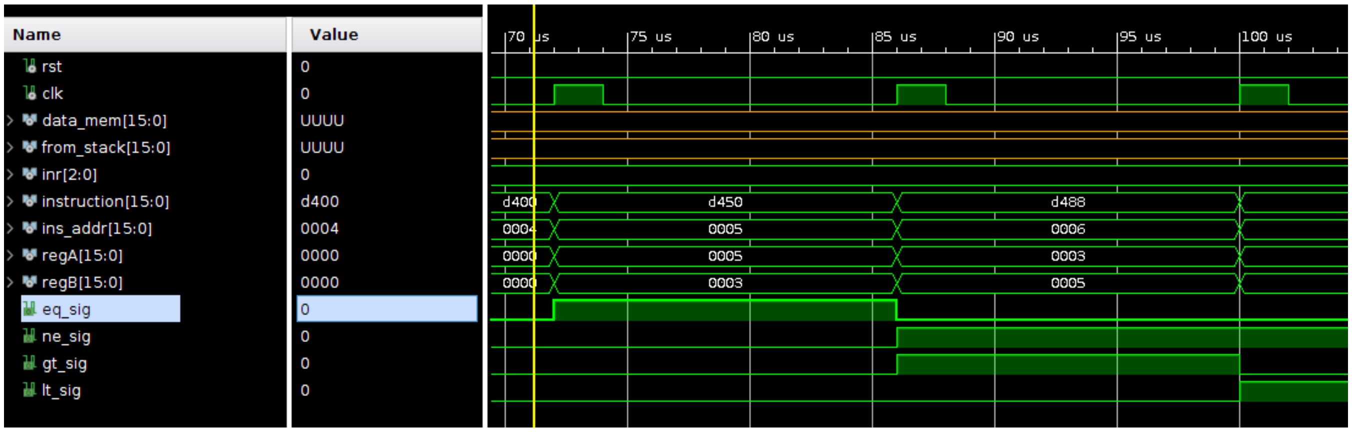
Figure 13: Lui instruction simulation

A simulation of the lui instruction is shown above in Figure 13. The Lui instruction loads the upper 6-bits of a register with a specified immediate value. Lui zeros the bottom 10-bits of the specified register. In the simulation the specified immediate value is 63 or binary 111111. After the clock edge it can be seen that lui correctly loaded the 6-bit immediate and cleared the bottom 10-bits of register one.

3 Branching Instruction Testing

3.1 CMP instruction verification

Instruction	Machine Code
cmp r0, r0	1101010000000000
cmp r1, r2	1101010001010000
cmp r2, r1	1101010010001000



Name	Value
rst	0
clk	0
data_mem[15:0]	UUUU
from_stack[15:0]	UUUU
inr[2:0]	0
instruction[15:0]	d400
ins_addr[15:0]	0004
regA[15:0]	0000
regB[15:0]	0000
eq_sig	0
ne_sig	0
gt_sig	0
lt_sig	0

Figure 14: Cmp instruction simulation

A simulation of the cmp instruction is shown above in Figure 14. Three cmp instructions are executed showing each possible output of the cmp flags (eq_sig, ne_sig, gr_sig, lt_sig). The cmp instruction is used before a conditional branch instruction to determine if the branch should be taken.

3.2 Beq instruction verification

Instruction	Machine Code
cmp r2, r1	1101010010001000
zero r4	1100100100100100
addi r4, 3	1101000100000011
bne 2	0010100000000010
zero r4	1100100100100100
blt 2	0011100000000010
zero r4	1100100100100100
zero r1	1100100001001001
zero r2	1100100010010010
cmp r1, r2	1101010001010000
beq 2	0010000000000010
zero r4	1100100100100100
zero r1	1100100001001001
zero r2	1100100010010010
addi r1, 1	1101000001000001
cmp r1, r2	1101010001010000
bgt 2	0011000000000010
zero r4	1100100100100100
hlt	0000000000000000

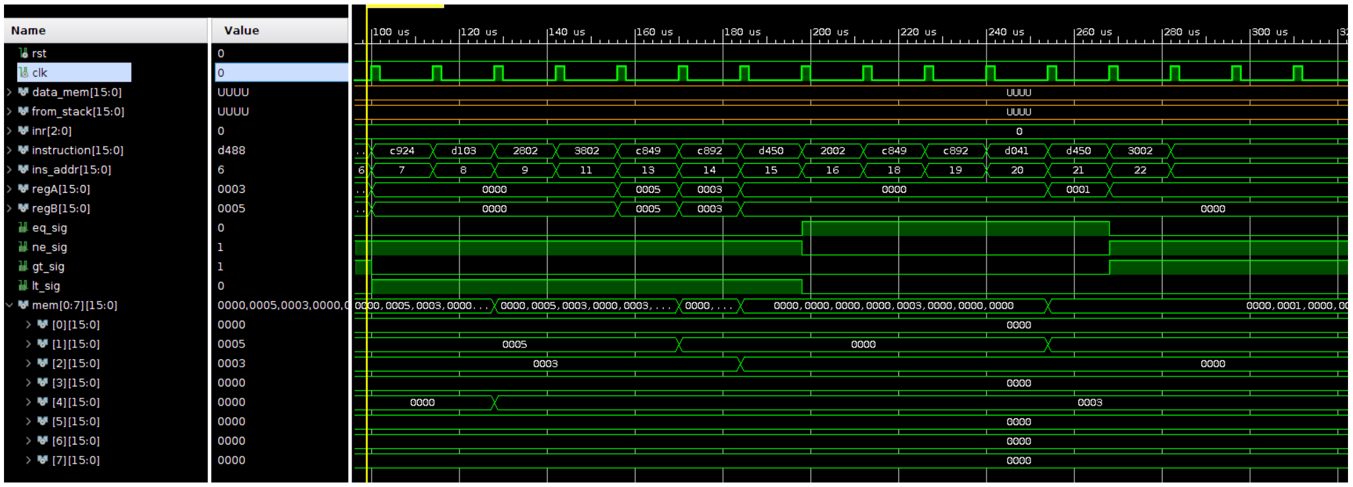


Figure 15: Beq instruction simulation

A simulation of the conditional branching instructions is shown above in Figure 15. Each conditional branch instruction is tested. The program gets to the end without clearing the value in r4 this shows that each of the four unique conditional branch instructions worked. In the above code zero is a pseudo-instruction that xors a register with itself.

3.3 Jalr instruction verification

Instruction	Machine Code
addi r1, 1	1101000001000001
jalr r3, r0	1101010001010000

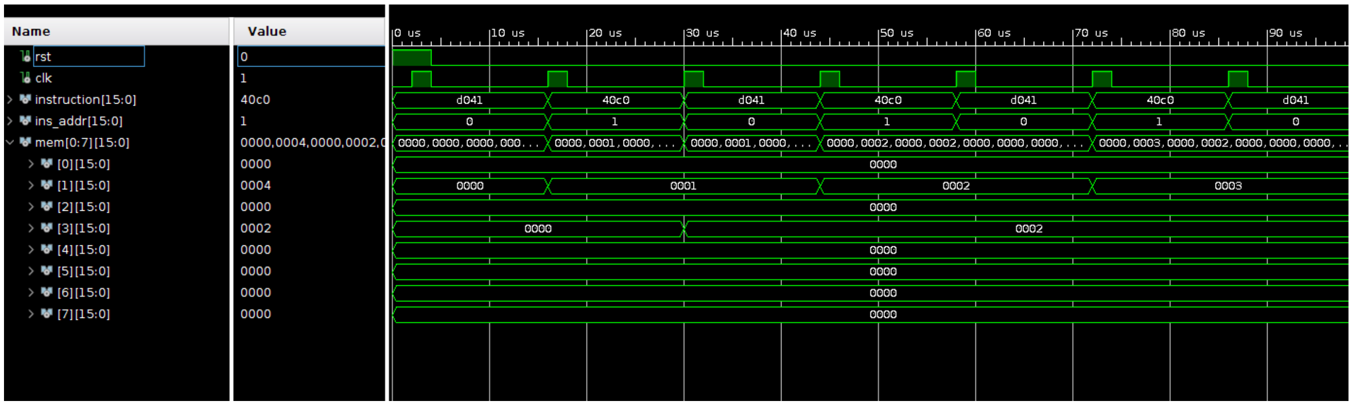
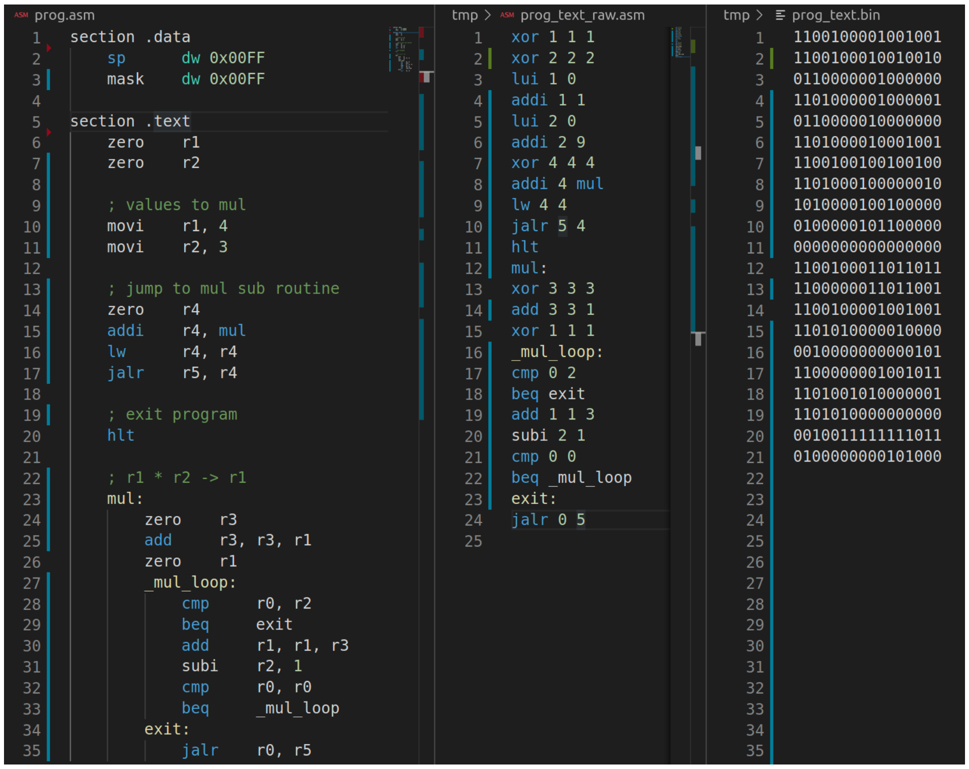


Figure 16: Jalr instruction simulation

A simulation of the jalr instruction is shown above in Figure 16. The jump and link register instruction works as indicated by the increasing values of register one. The simulation also shows the correct return address loaded into register three.

4 CPU Test Program



```
section .data
    sp      dw 0x00FF
    mask    dw 0x00FF

section .text
    zero    r1
    zero    r2

    ; values to mul
    movi    r1, 4
    movi    r2, 3

    ; jump to mul sub routine
    zero    r4
    addi    r4, mul
    lw      r4, r4
    jalr    r5, r4

    ; exit program
    hlt

    ; r1 * r2 -> r1
    mul:
        zero    r3
        add     r3, r3, r1
        zero    r1
        _mul_loop:
            cmp     r0, r2
            beq     exit
            add     r1, r1, r3
            subi    r2, 1
            cmp     r0, r0
            beq     _mul_loop
        exit:
            jalr    r0, r5
```

```
xor 1 1 1
xor 2 2 2
lui 1 0
addi 1 1
lui 2 0
addi 2 9
xor 4 4 4
addi 4 mul
lw 4 4
jalr 5 4
hlt
mul:
xor 3 3 3
add 3 3 1
xor 1 1 1
_mul_loop:
cmp 0 2
beq exit
add 1 1 3
subi 2 1
cmp 0 0
beq _mul_loop
exit:
jalr 0 5
```

```
1100100001001001
1100100010010010
0110000001000000
1101000001000001
0110000010000000
1101000010001001
1100100100100100
1101000100000010
1010000100100000
0100000101100000
0000000000000000
1100100011011011
1100000011011001
1100100001001001
1101010000010000
0010000000000101
1100000001001011
1101001010000001
1101010000000000
0010011111111011
0100000000101000
```

Figure 17: Software Multiplication Test Program

A test program is created to further verify the functionality of the Shell CPU. An obvious short coming of the Shell ISA is that it provides no multiplication instruction. To remedy the lack of multiplication hardware support the test program illustrates how multiplication can be performed in software Figure 17. Three code segments can be seen in Figure 17. The first code segment (far left) is the high level assembly code for the program. The middle code segment contains the bare metal assembly code (no pseudo-instructions or register names). The last code segment shows the machine code that is loaded onto the processor to perform the multiplication. It is important to note that this code not only performs multiplication but does so in a reusable way with subroutines.

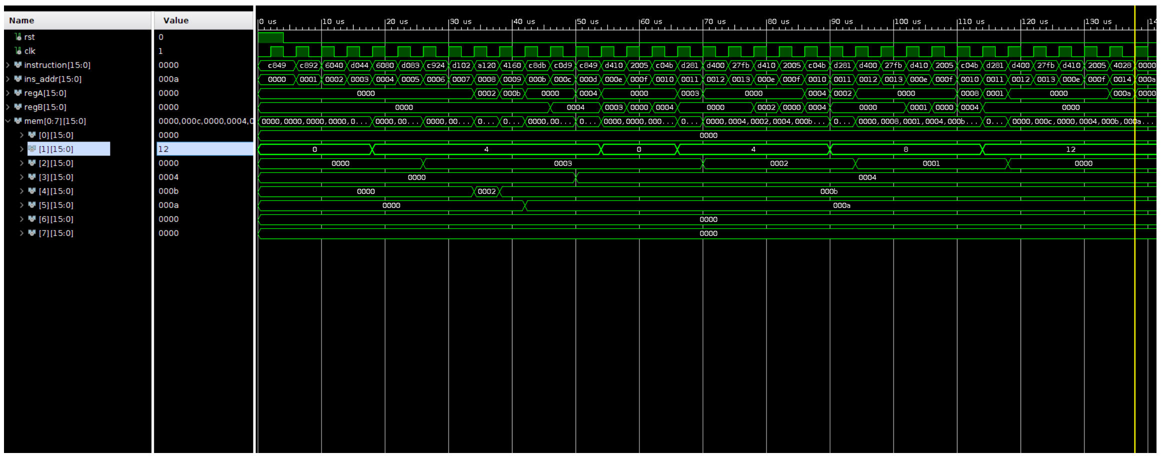


Figure 18: Software Multiplication Test Program Simulation

The simulation of the test program from Figure 17 can be seen in Figure 18. In the test program the values three and four were passed into the multiplication subroutine. The subroutine is expected to compute the result and return to the caller. In the simulation it can be seen that twelve, the answer to three times four, is stored in register one. The program then takes a few clock cycles to return from the multiplication subroutine where it then executes a halt instruction.