Advanced Digital Logic

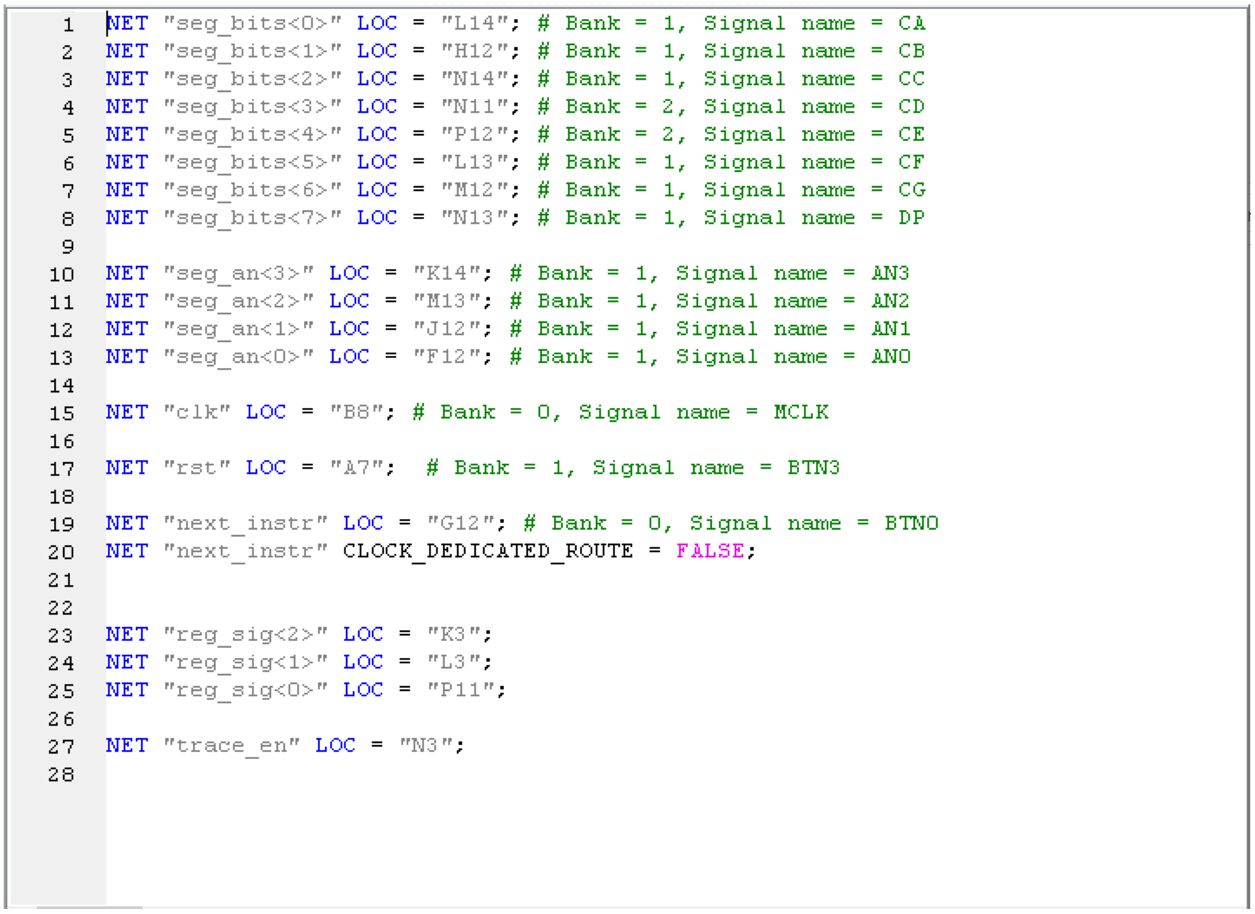
ENGR – UH 2310

Spring 2025

Instructor: Muhammad Hassan Jamil

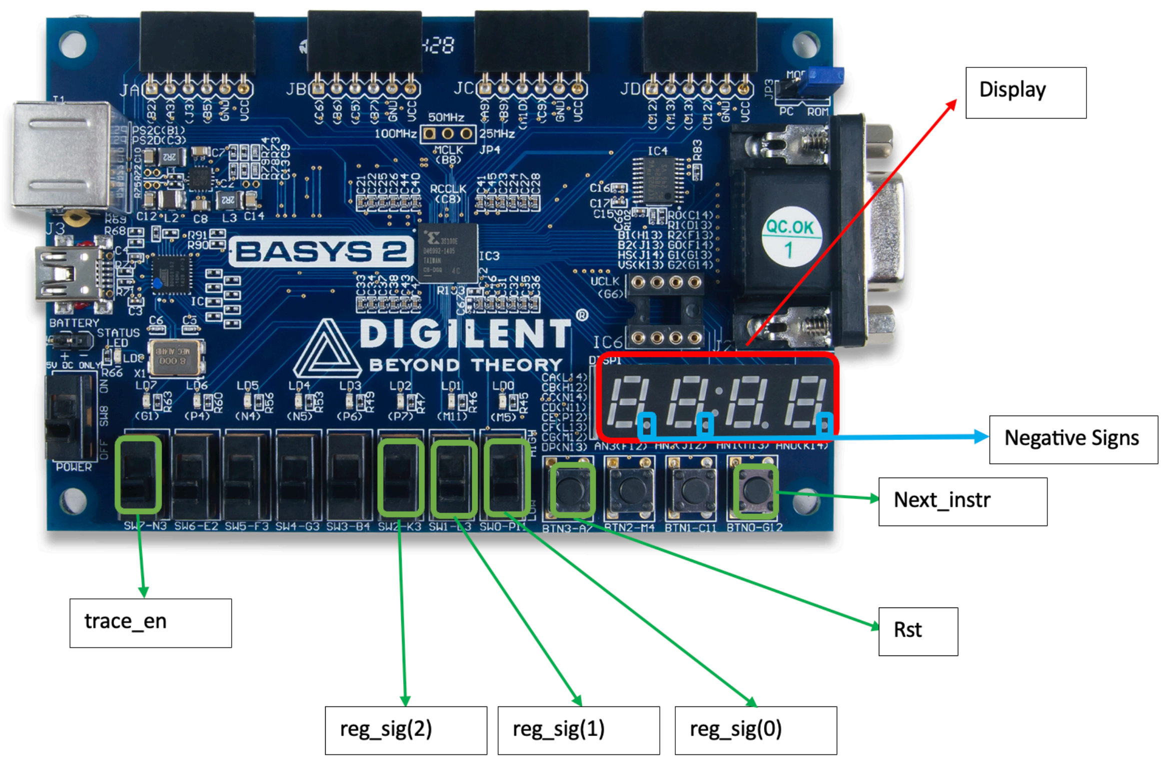
Demarce Williams (dsw9740)
Aman Sunesh (as18181)

UCF WITH SCHEMATIC (CONTRAINT FLLE):



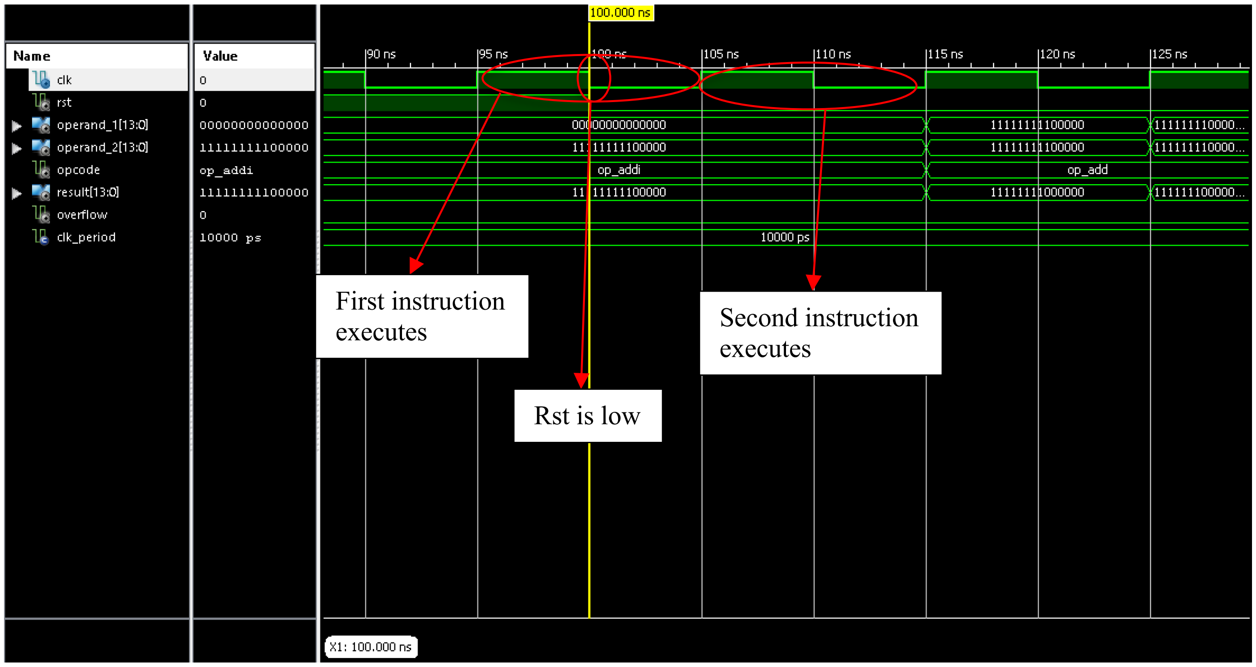
```
NET "seg_bits<0>" LOC = "L14"; # Bank = 1, Signal name = CA
NET "seg_bits<1>" LOC = "H12"; # Bank = 1, Signal name = CB
NET "seg_bits<2>" LOC = "N14"; # Bank = 1, Signal name = CC
NET "seg_bits<3>" LOC = "N11"; # Bank = 2, Signal name = CD
NET "seg_bits<4>" LOC = "P12"; # Bank = 2, Signal name = CE
NET "seg_bits<5>" LOC = "L13"; # Bank = 1, Signal name = CF
NET "seg_bits<6>" LOC = "M12"; # Bank = 1, Signal name = CG
NET "seg_bits<7>" LOC = "N13"; # Bank = 1, Signal name = DP

NET "seg_an<3>" LOC = "K14"; # Bank = 1, Signal name = AN3
NET "seg_an<2>" LOC = "M13"; # Bank = 1, Signal name = AN2
NET "seg_an<1>" LOC = "J12"; # Bank = 1, Signal name = AN1
NET "seg_an<0>" LOC = "F12"; # Bank = 1, Signal name = AN0

NET "clk" LOC = "B8"; # Bank = 0, Signal name = MCLK

NET "rst" LOC = "A7";  # Bank = 1, Signal name = BTN3

NET "next_instr" LOC = "G12"; # Bank = 0, Signal name = BTN0
NET "next_instr" CLOCK_DEDICATED_ROUTE = FALSE;


NET "reg_sig<2>" LOC = "K3";
NET "reg_sig<1>" LOC = "L3";
NET "reg_sig<0>" LOC = "P11";

NET "trace_en" LOC = "N3";

```

LABELLED HD PICTURE OF INPUTS AND OUTPUTS ON FPGA BOARD FOR MPU



Please note that the register was slightly modified to implement the trace operation.

Simulation Snapshots
ADDI R1, R0, -32 // R1 = R0 + (-32) = -32
ADDI R2, R0, -32 // R2 = R0 + (-32) = -32



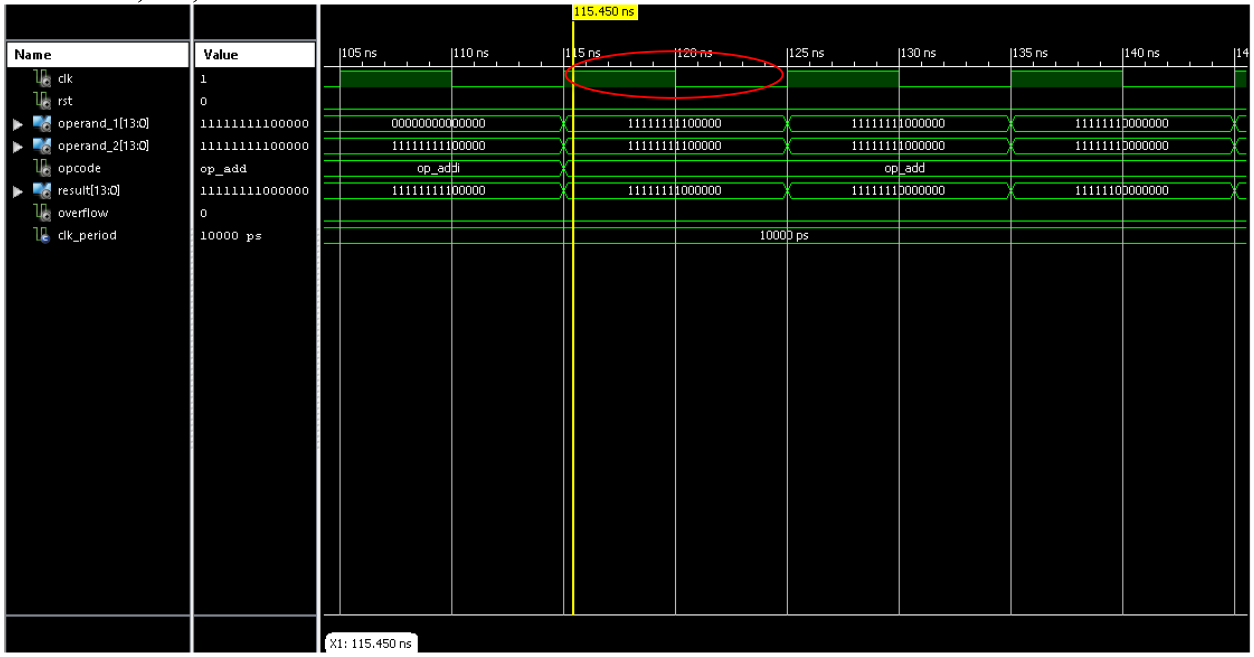
The first ADDI instruction starts execution at 95 ns, just before reset is set to low at 100 ns, completing half a cycle by then. After 105 ns the first instruction is executed completely, the second ADDI instruction executes in the next full cycle. This results in both instructions together appearing to take 1.5 cycles to complete while reset is low.

However each instruction took 1 cycle (10 ns) to complete.

Note also that there is no visible change in the two instructions present as they are the same, but with different destination registers.

Instruction one adds 0 and -32 and stores the result of -32 in Register 1.
Instruction one adds 0 and -32 and stores the result of -32 in Register 2.

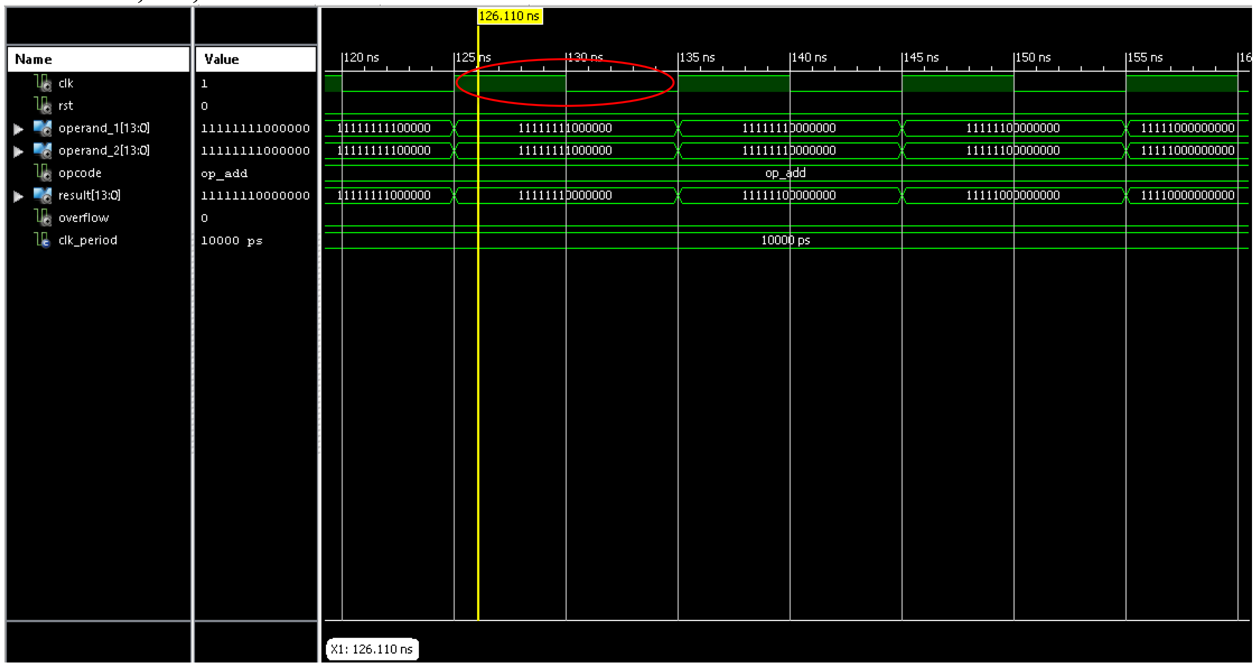
ADD   R3, R1, R2   // R3 = R1 + R2 = -64



Name	Value
clk	1
rst	0
operand_1[13:0]	11111111100000
operand_2[13:0]	11111111100000
opcode	op_add
result[13:0]	11111111000000
overflow	0
clk_period	10000 ps

This instruction adds -32 and -32 from the previous Registers R1 and R2 respectively, and store the result -64 in Register R3.
The instruction took 1 cycle (10 ns) to complete.

ADD   R3, R3, R3   // R3 = R3 + R3 = -128

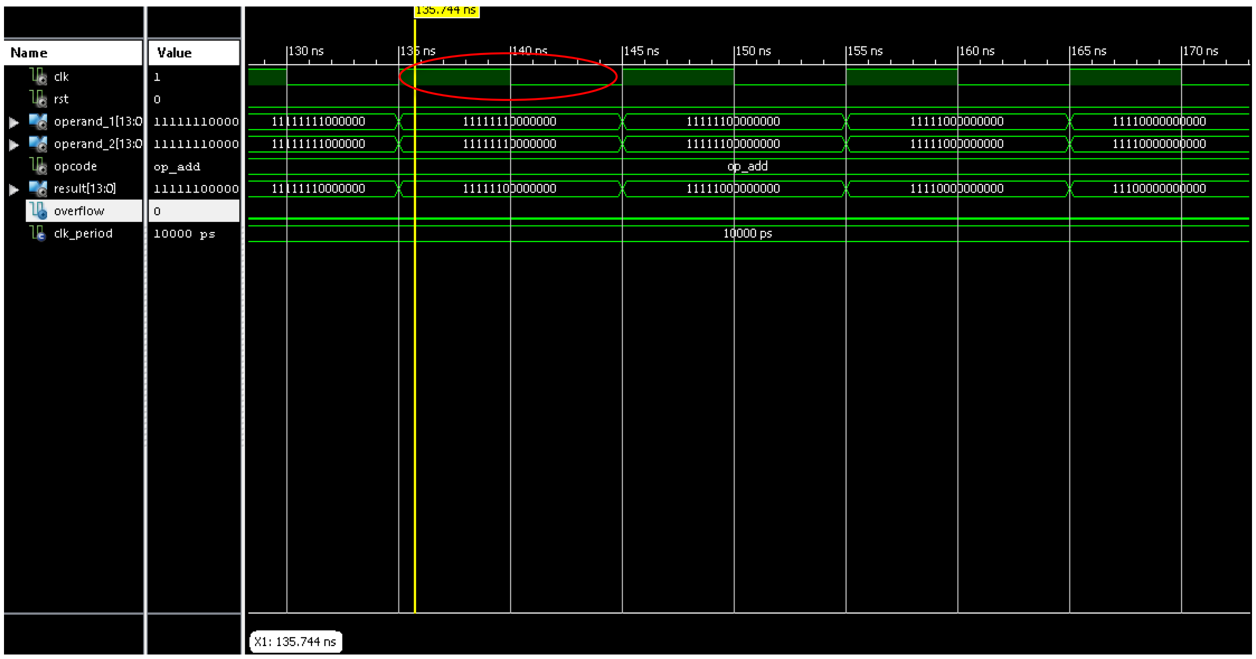


Name	Value
clk	1
rst	0
operand_1[13:0]	11111111000000
operand_2[13:0]	11111111000000
opcode	op_add
result[13:0]	11111110000000
overflow	0
clk_period	10000 ps

```
operand_1: 11111111100000 | 11111111000000 | 11111110000000 | 11111100000000 | 11111000000000
operand_2: 11111111100000 | 11111111000000 | 11111110000000 | 11111100000000 | 11111000000000
opcode:    op_add
result:    11111111000000 | 11111110000000 | 11111100000000 | 11111000000000 | 11110000000000
clk_period: 10000 ps
```

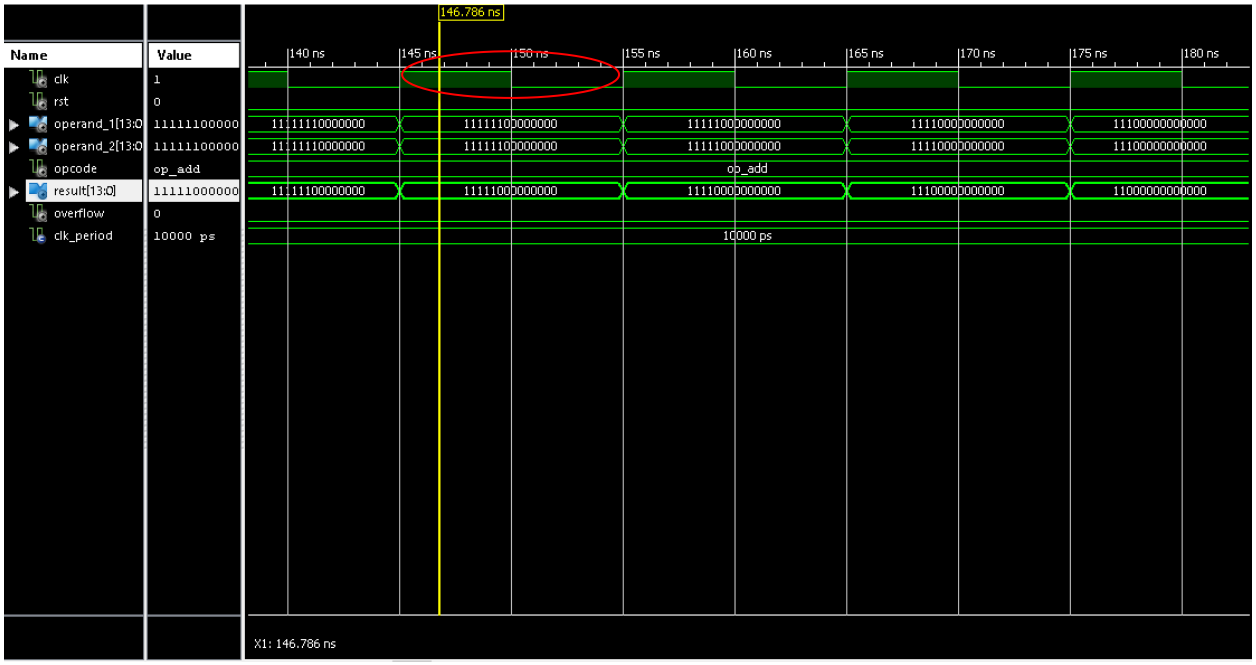
This instruction adds the result -64 from the previous instruction stored in R3 to itself, and store the result -128 in Register R3.
The instruction took 1 cycle (10 ns) to complete.

ADD   R3, R3, R3   // R3 = R3 + R3 = -256



This instruction adds the result -128 from the previous instruction stored in R3 to itself, and store the result -256 in Register R3.
The instruction took 1 cycle (10 ns) to complete.

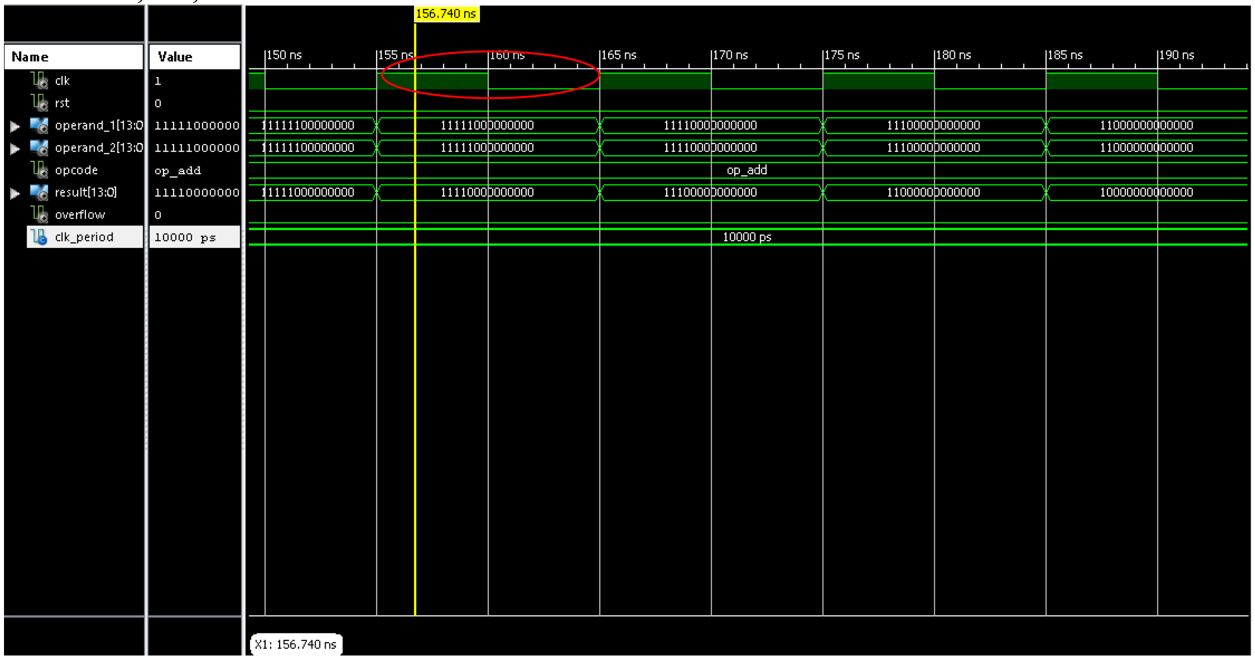
ADD   R3, R3, R3   // R3 = R3 + R3 = -512



Name	Value
clk	1
rst	0
operand_1[13:0	11111100000
operand_2[13:0	11111100000
opcode	op_add
result[13:0]	11111000000
overflow	0
clk_period	10000 ps

This instruction adds the result -256 from the previous instruction stored in R3 to itself, and store the result -512 in Register R3.
The instruction took 1 cycle (10 ns) to complete.

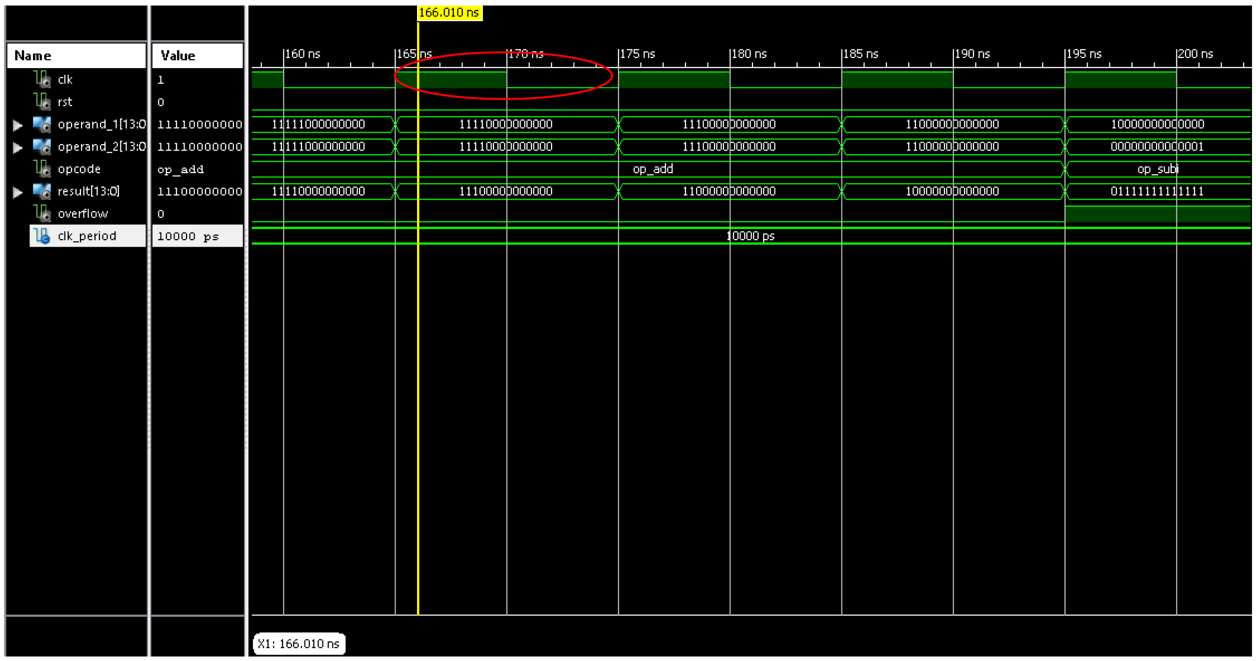
ADD   R3, R3, R3   // R3 = R3 + R3 = -1024



Name	Value
clk	1
rst	0
operand_1[13:0]	11111000000
operand_2[13:0]	11111000000
opcode	op_add
result[13:0]	11110000000
overflow	0
clk_period	10000 ps

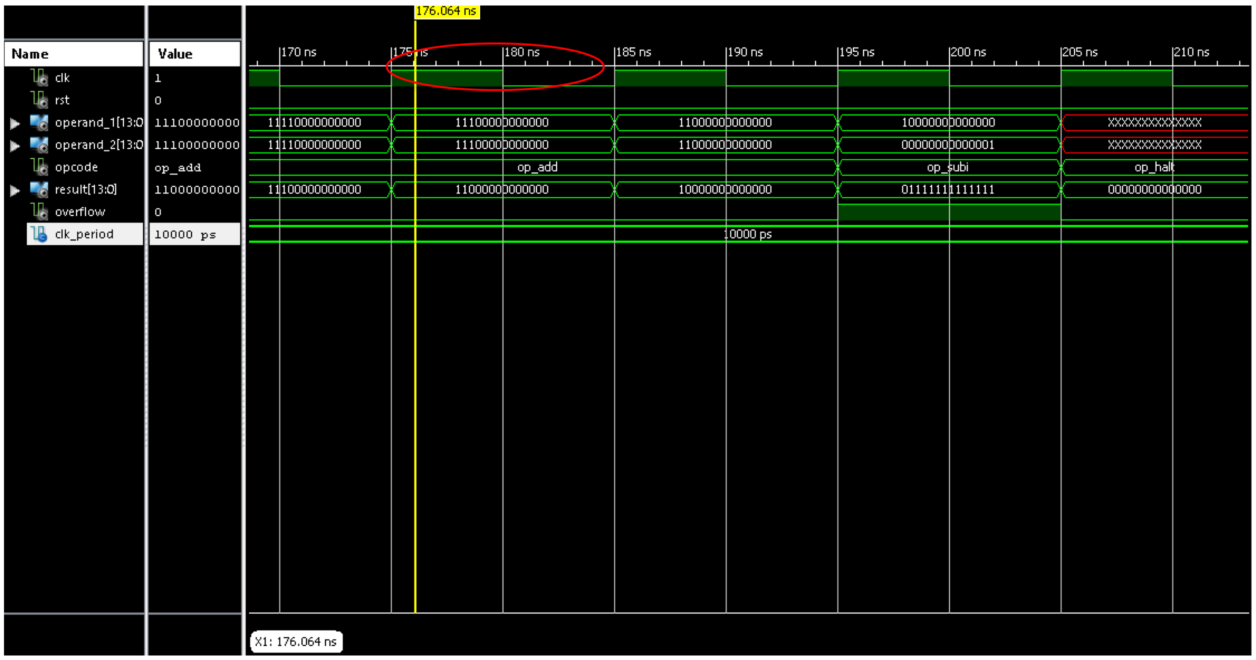
This instruction adds the result -512 from the previous instruction stored in R3 to itself, and store the result -1024 in Register R3.
The instruction took 1 cycle (10 ns) to complete.

ADD R3, R3, R3 // R3 = R3 + R3 = -2048



This instruction adds the result -1024 from the previous instruction stored in R3 to itself, and store the result -2048 in Register R3.
The instruction took 1 cycle (10 ns) to complete.

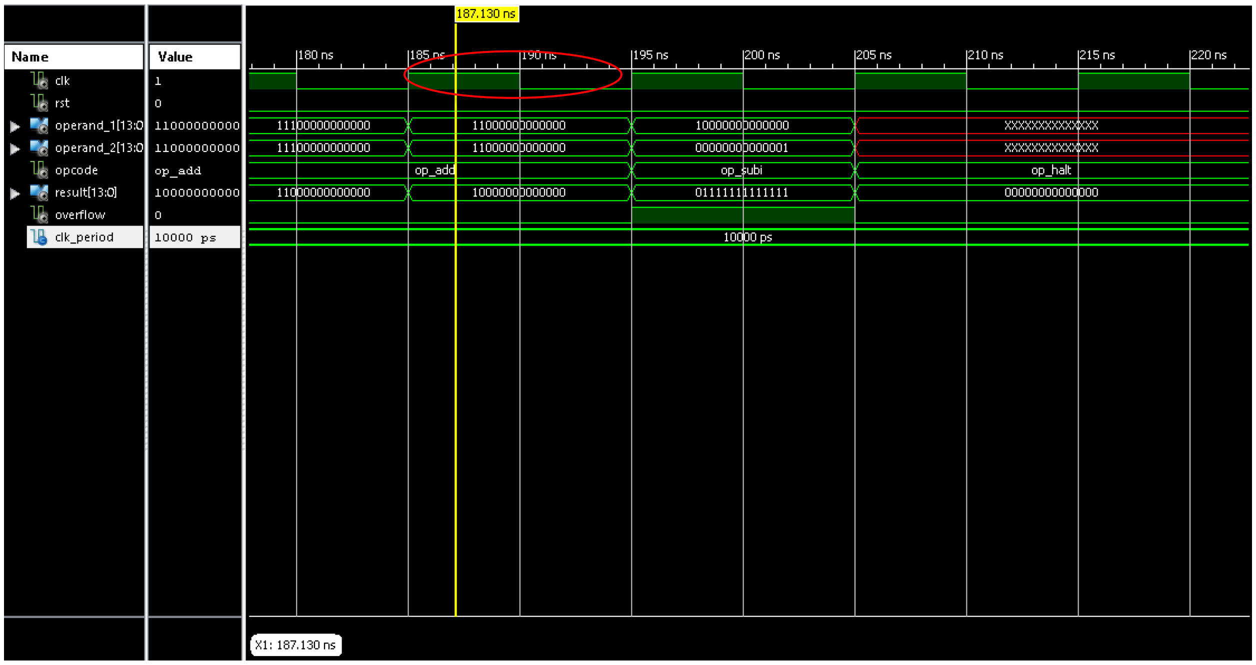
ADD R3, R3, R3 // R3 = R3 + R3 = -4096



This instruction adds the result -2048 from the previous instruction stored in R3 to itself, and store the result -4096 in Register R3.

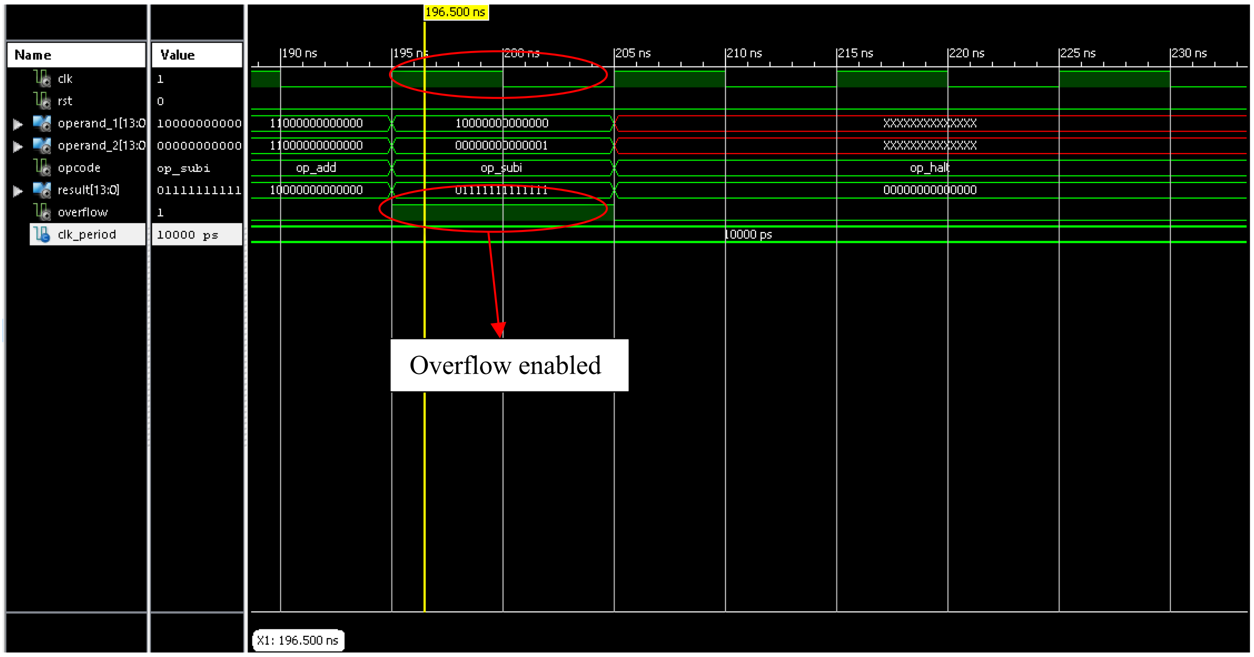
The instruction took 1 cycle (10 ns) to complete.

ADD   R3, R3, R3   // R3 = R3 + R3 = -8192



This instruction adds the result -4096 from the previous instruction stored in R3 to itself, and store the result -8192 in Register R3.
The instruction took 1 cycle (10 ns) to complete.

SUBI R4, R3, -1   // R4 = R3 - (-1) = -8193 (overflow, should be 8191)

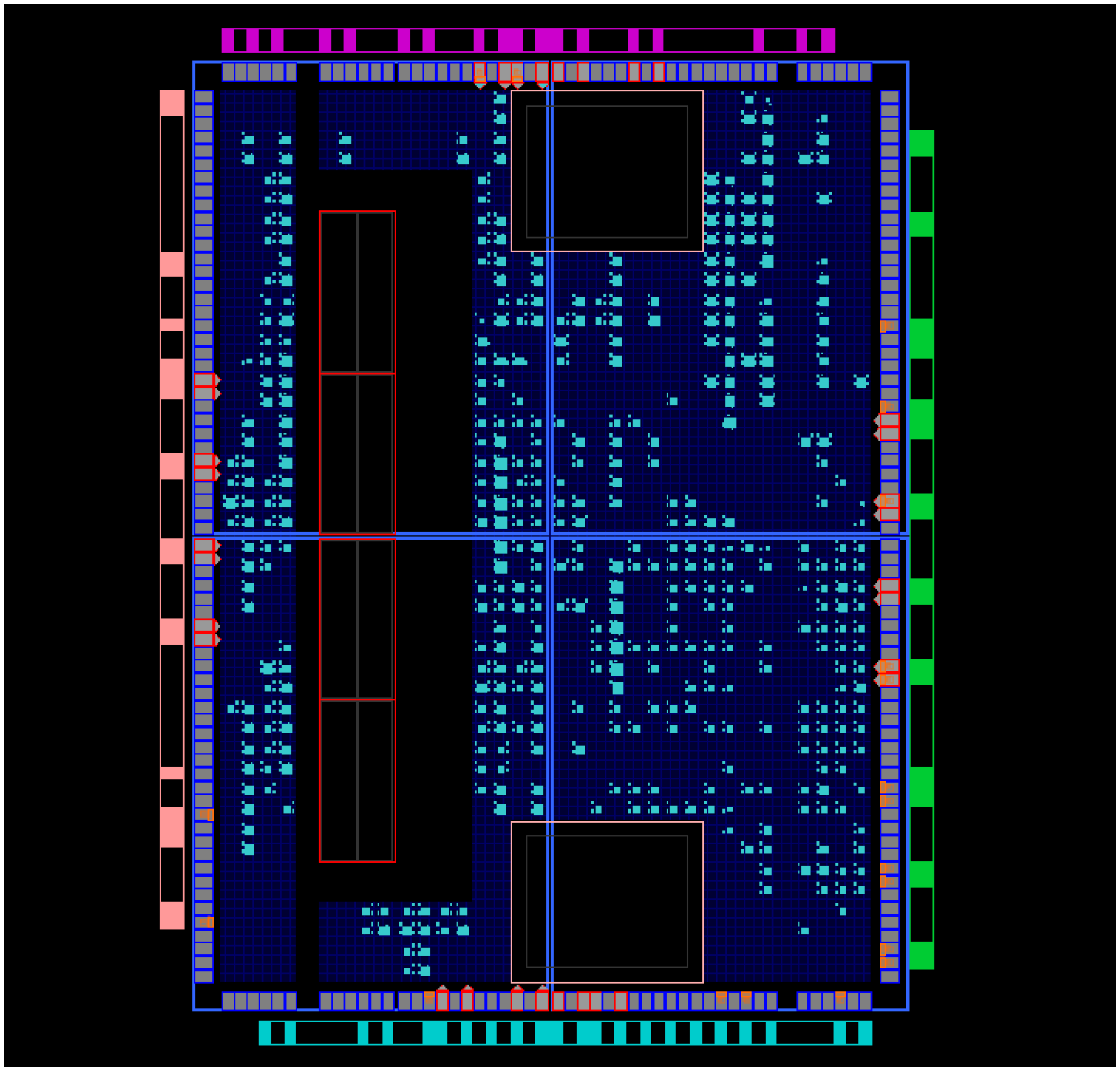


This instruction subtracts the immediate value -1 from the result -8192 from previous instruction stored in R3, and stores the result in Register R3.
The expected result is -8191, but this operation results in a overflow case and produces 8192 instead. Therefore the overflow signal is enabled.

The instruction took 1 cycle (10 ns) to complete.

From the simulations it can be concluded that one instruction. takes one (1) clock cycle to fully execute.

Floor Plan Design (PlanAhead)



Decoded Task 3 Instructions

Instruction: 1001 001 000 000 101

Opcode: 1001 → **ADDI** (Add Immediate)

Destination Register (Rd): 001 → **R1**

Source Register (Rs1): 000 → **R0**

Immediate: 000101 → **5** (decimal)

Operation: R1 = R0 + 5

Since R0 is assumed to be 0, **R1 becomes 5.**

Instruction: 1001 010 000 000 000

Opcode: 1001 → **ADDI**

Destination Register (Rd): 010 → **R2**

Source Register (Rs1): 000 → **R0**

Immediate: 000000 → **0** (decimal)

Operation: R2 = R0 + 0

So, **R2 remains 0.**

Instruction: 1001 010 010 000 101

Opcode: 1001 → **ADDI**

Destination Register (Rd): 010 → **R2**

Source Register (Rs1): 010 → **R2**

Immediate: 000101 → **5** (decimal)

Operation: R2 = R2 + 5

Since R2 was 0 from Instruction 2, **R2 becomes 5.**

Instruction: 1011 001 001 000 001

Opcode: 1011 → **SUBI** (Subtract Immediate)

Destination Register (Rd): 001 → **R1**

Source Register (Rs1): 001 → **R1**

Immediate: 000001 → **1** (decimal)

Operation: R1 = R1 – 1

Since R1 was 5 (from Instruction 1), **R1 becomes 4.**

Instruction: 1100 111 000 001 110

Opcode: 1100 → **BLT** (Branch if Less Than)

Source Register (Rs1): 000→ R0

Source Register (Rs2): 001→ R1

Fields:

> The typical 3 bits for the destination register "111" serve as part of the branch offset (or sign-extension field) and here indicate register R7 is involved for the immediate value calculation.
>
> Registers R0 and R1 are used in the comparison.

Immediate: 001110 → **14** (decimal)

Operation: For BLT, the processor checks if (R0 < R1). By convention, R0 is always 0 and R1 is 4 from instruction 4. So the condition is true. The PC is incremented by the immediate value of 14. So the program counter will point to the 18th instruction.

Final Register Values:
R1 = 4, R2 = 5, R0 = R7 = 0

Implementation of the Fibonacci Sequence Algorithm

The **Fibonacci sequence** is a series where each term is the sum of the two preceding terms, starting with **0 and 1**:

$$F(n)=F(n-1)+F(n-2)$$

where $F(0)=0$ and $F(1)=1$.

In the algorithm, the **ADDI, ADD, BLT (Branch if Less Than), JUMP, and HALT** operations are used to compute the **nth** Fibonacci term:

- **ADDI**:
  - Initializes the first two Fibonacci terms, **0 and 1**, in registers **R2** and **R3**, respectively.
  - Stores **n** (the desired Fibonacci term index) in **R1** and **1** in **R7** (used as a loop termination condition).
  - Swaps register values to ensure **R2 and R3** always hold the two most recent terms of the sequence.
- **BLT (Branch if Less Than)**:
  - Checks if **R1 < R7**; if true, the loop terminates and **HALT** is executed.
  - Otherwise, the loop continues until the **nth** term is computed.
- **ADD**:
  - Computes the next Fibonacci term by adding **R2 and R3**, storing the result in **R4**.
- **JUMP**:
  - Loops back **five instructions** to re-execute **BLT**, allowing Fibonacci terms to be computed iteratively.
- **HALT**:
  - Terminates execution once the **nth** Fibonacci term is reached.

The instructions used to implement this algorithm are as follows:

Instruction 0 (Load 1 into R7):
```
1001 111 000 000 001
```
(ADDI: R7 = R0 + 1)

Instruction 1 (Load 6 into R1):
```
1001 001 000 000 110
```
(ADDI: R1 = R0 + 6)

Instruction 2 (Initialize R2 to 0):
```
1001 010 000 000 000
```
(ADDI: R2 = R0 + 0)

Instruction 3 (Initialize R3 to 1):
```
1001 011 000 000001
```
(ADDI: R3 = R0 + 1)

Instruction 4 (Branch if R1 < R7):
```
1100 000 001 111 110
```
(BLT: if R1 < R7, then PC = PC + IV; here the immediate is derived from the Rd and tail fields)

Instruction 5 (Calculate next Fibonacci term: R4 = R2 + R3):
```
1000 100 010 011 000
```
(ADD: R4 = R2 + R3)

Instruction 6 (Move R3 into R2):
```
1001 010 011 000 000
```
(ADDI: R2 = R3 + 0)

Instruction 7 (Move R4 into R3):
```
1001 011 100 000 000
```
(ADDI: R3 = R4 + 0)

Instruction 8 (Decrement n in R1):
```
1011 001 001 000 001
```
(SUBI: R1 = R1 – 1)

Instruction 9 (Jump back to instruction 4):
```
1110 000 000 111 011
```
(JMP: PC = PC + (-5); the immediate "111011" represents –5 in two’s complement)

Instruction 10 (Halt the processor):
```
1111 000 000 000 000
```
(HALT)

Program Execution

Initialization:

R7 = 1

R1 = 6 (input n; 6 iterations will compute the 7th Fibonacci number)

R2 = 0 (F(0) = 0)

R3 = 1 (F(1) = 1)

Iteration 1:

BLT: Check if R1 (6) < R7 (1) → false

ADD: R4 = R2 + R3 = 0 + 1 = 1

Move: R2 = R3 → R2 becomes 1

Move: R3 = R4 → R3 becomes 1

SUBI: R1 = R1 – 1 → R1 becomes 5

JMP: Loop back

Iteration 2:

BLT: Check if R1 (5) < R7 (1) → false

ADD: R4 = R2 + R3 = 1 + 1 = 2

Move: R2 = R3 → R2 becomes 1

Move: R3 = R4 → R3 becomes 2

SUBI: R1 = 5 – 1 → R1 becomes 4

JMP: Loop back

Iteration 3:

BLT: Check if R1 (4) < R7 (1) → false

ADD: R4 = R2 + R3 = 1 + 2 = 3

Move: R2 = R3 → R2 becomes 2

Move: R3 = R4 → R3 becomes 3

SUBI: R1 = 4 – 1 → R1 becomes 3

JMP: Loop back

Iteration 4:

BLT: Check if R1 (3) < R7 (1) → false

ADD: R4 = R2 + R3 = 2 + 3 = 5

Move: R2 = R3 → R2 becomes 3

Move: R3 = R4 → R3 becomes 5

SUBI: R1 = 3 – 1 → R1 becomes 2

JMP: Loop back

Iteration 5:

BLT: Check if R1 (2) < R7 (1) → false

ADD: R4 = R2 + R3 = 3 + 5 = 8

Move: R2 = R3 → R2 becomes 5

Move: R3 = R4 → R3 becomes 8

SUBI: R1 = 2 – 1 → R1 becomes 1

JMP: Loop back

Iteration 6:

BLT: Check if R1 (1) < R7 (1) → false (since 1 is not less than 1)

ADD: R4 = R2 + R3 = 5 + 8 = 13

Move: R2 = R3 → R2 becomes 8

Move: R3 = R4 → R3 becomes 13

SUBI: R1 = 1 – 1 → R1 becomes 0

JMP: Loop back

Iteration 7 (Exit):

BLT: Check if R1 (0) < R7 (1) → true

Branch to HALT instruction

Final Results:

R1 = 0, R2 = 8, R3 = 13, R4 = 13, R7 = 1

The computed 7th Fibonacci number (F(7)) is 13 (stored in R3)