Design and Implementation of a 16-bit Single-Cycle RISC CPU Using Logisim-Evolution 3.9.0

King Fahd University of Petroleum and Minerals

COE 301: Computer Architecture & Assembly Language (Term 242)

10th May 2025

Name: Amer Almutairi

ID: 202169770

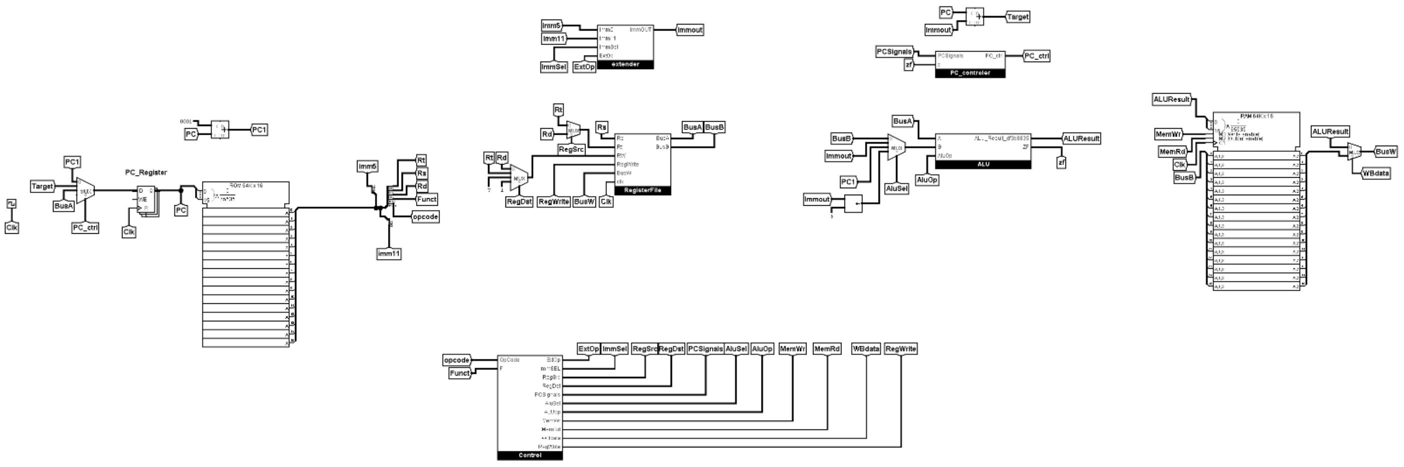
Section#: 53, Monday

Abstract

This report describes the design, implementation, and verification of a 16-bit, five-stage pipelined RISC processor using Logisim-evolution . The processor implements a small RISC-style ISA with R-type, I-type, and J-type instructions, including arithmetic, logic, memory, branch, and bit-reversal operations. We detail the Datapath, control logic, forwarding and hazard-detection units, test programs, and team organization.

Contents

1 Datapath Components

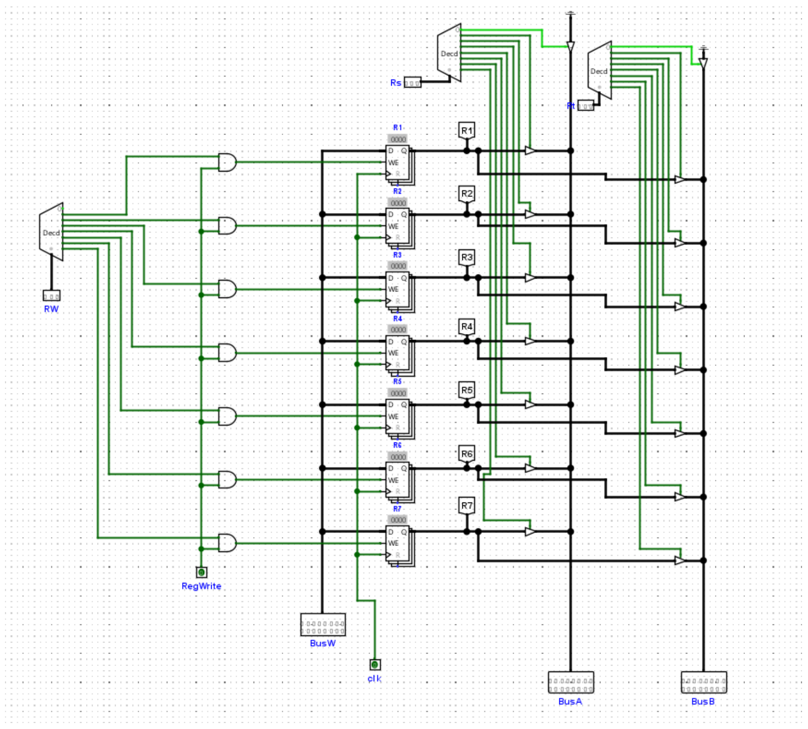


1.1 Register File

R1–R7 (16 bits each), R0 hardwired to 0.

On the rising edge of Clock, if RegWrite = 1, the 16-bit WriteData is written into the register indexed by RW.

At all times, the contents of registers RS and RT appear on BusA and BusB, respectively.



Input:

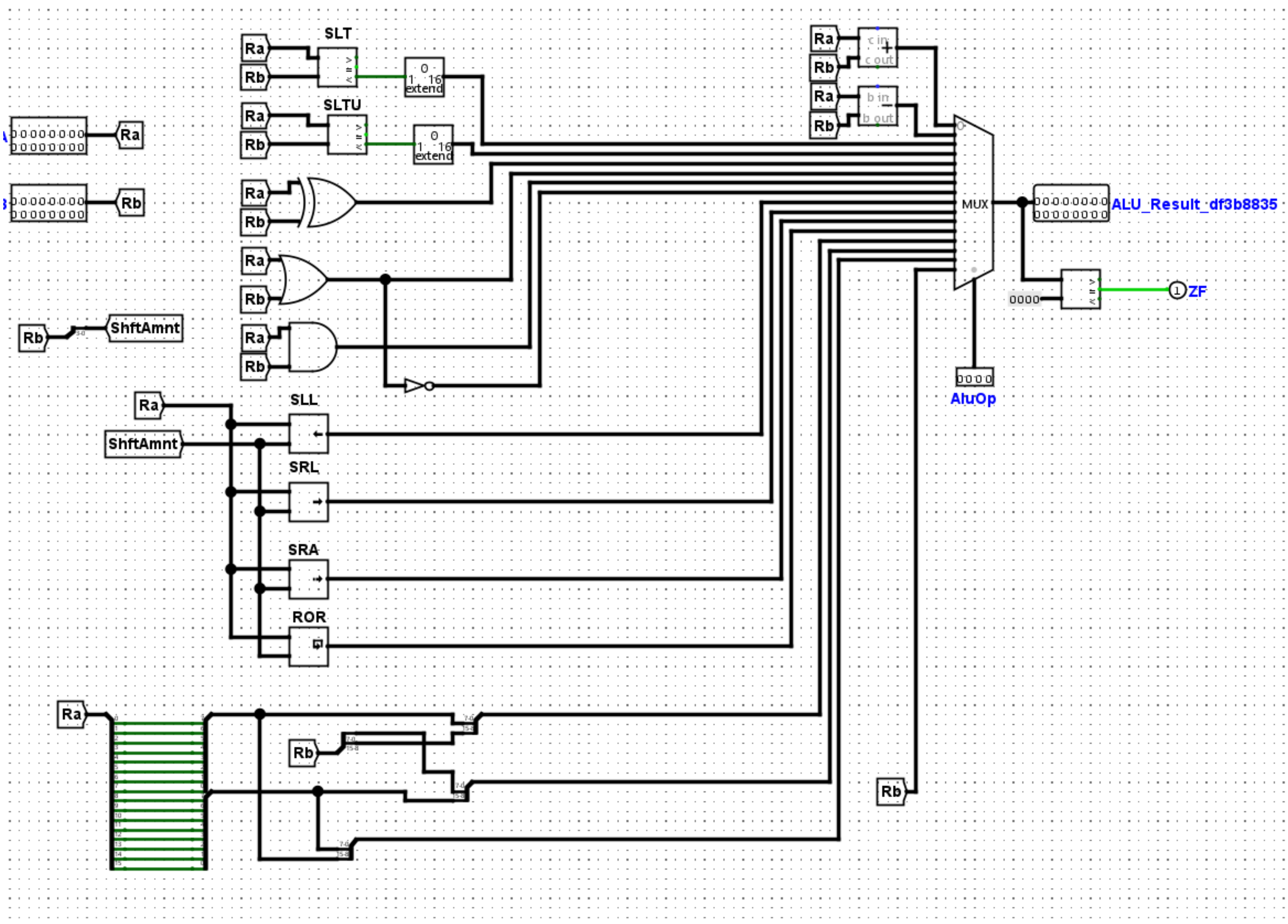
- **RS** (3 bits): address of source register A
- **RT** (3 bits): address of source register B
- **RW** (3 bits): address of destination registers
- **WriteData** (16 bits): data to be written
- **RegWrite** (1 bit): write-enable control

Output:

BusA (16 bits): data read from RS

BusB (16 bits): data read from RT

1.2 Arithmetic & Logic Unit (ALU)



Input:

- A (16 bits): operand A from BusA
- B (16 bits): operand B or immediate (via ALUSrc)
- ALUOp (4 bits): selects operation

Output:

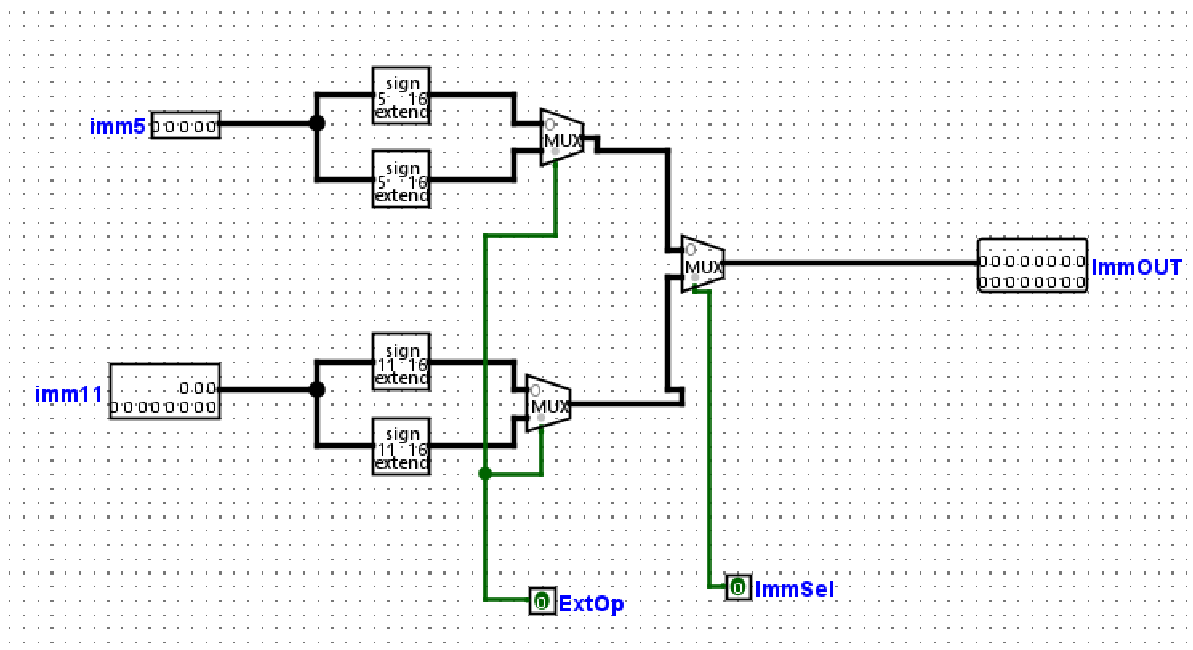
- ALUResult (16 bits): result

- Zero (1 bit): (ALUResult==0) flag

Supports the 16-bit operations listed in the spec, including:

- **Arithmetic:** ADD, SUB
- **Comparisons:** SLT (signed), SLTU (unsigned)
- **Bitwise:** AND, OR, XOR, NOR
- **Shifts/Rotates:** SLL, SRL, SRA, ROR
- **Reversal:** REVA, REVL, REVH

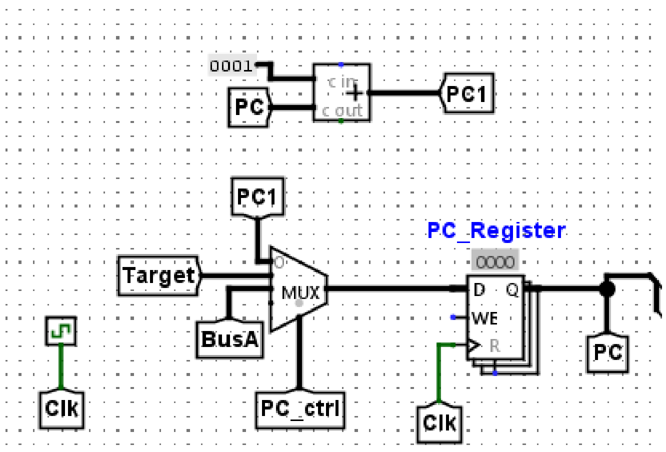
1.3 Sign/Zero Extender

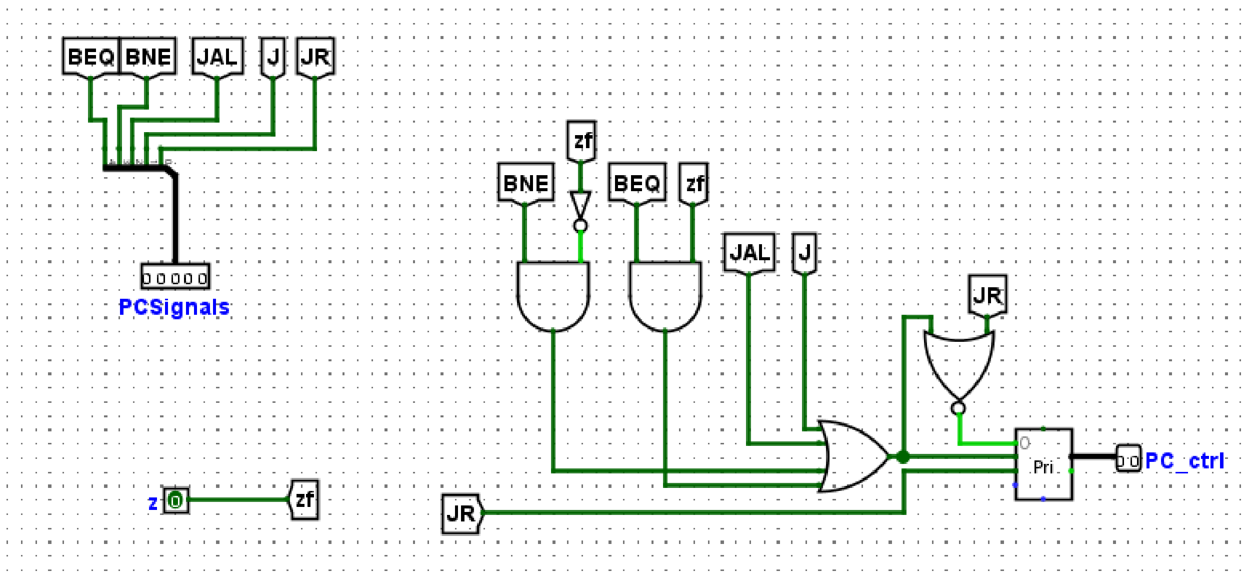


I-type: Sign-extends 5-bit immediates for most instructions; zero-extends for ANDI/ORI/XORI.

J-type: Zero-extends 11-bit immediates for J, JAL, and LUI.

1.4 Program Counter (PC) Logic





Computes NextPC:

PC+1 adder for sequential fetch.

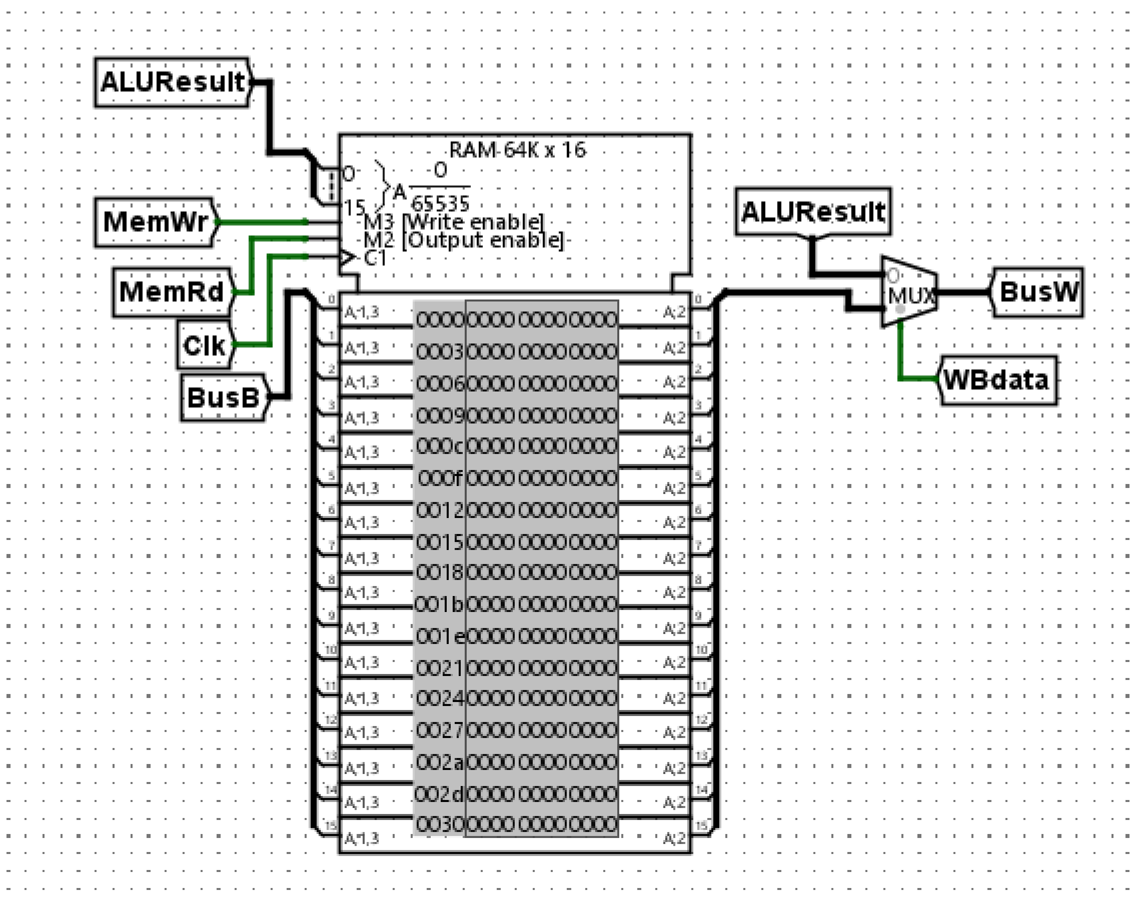
Branch adder: PC_in + sign_extend(Imm5) for BEQ/BNE.

Jump adder: PC_in + zero_extend(Imm11) for J/JAL.

MUX network selects next PC based on Branch & Zero, Jump, or fall-through

1.5 Data Memory

Implements a 64 K × 16-bit RAM. On the rising edge of Clk, if MemWr=1 it writes WriteData into the cell indexed by Addr. If MemRd=1, the contents at Addr drive ReadData, which is then selected by the WB-stage MUX as WBdata when appropriate.



Inputs:

- Addr (16 bits): address from ALUResult

- WriteData (16 bits): data from BusB

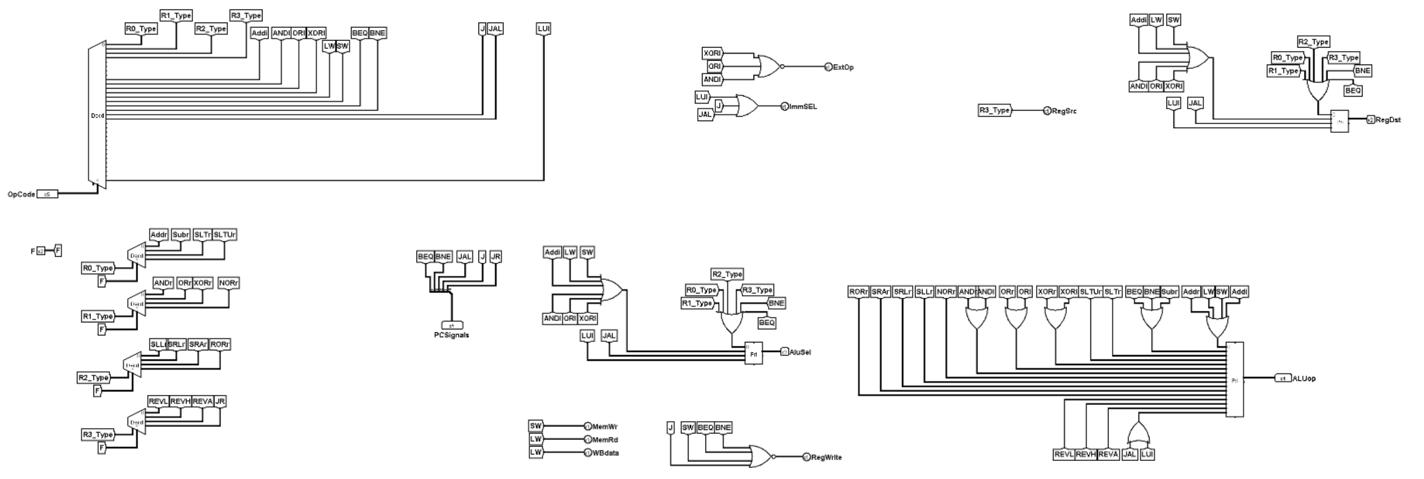
- MemWr (1 bit): memory-write enable

- MemRd (1 bit): memory-read enable

Outputs:

- ReadData (16 bits): data output when MemRd=1 (goes to BusW via MUX)

2 Control Logic



2.1 Control Table

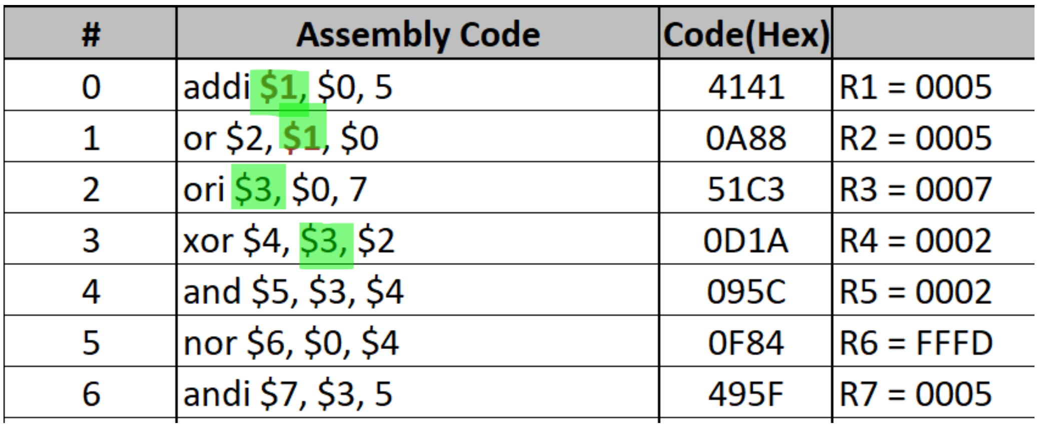
INSTR.	OP[4:0]	F[1:0]	REGDST	ALUSRC	MEMREAD	MEMWRITE	MEMTOREG	REGWRITE	BRANCH	JUMP	ALUOP
ADD	00000	00	1	0	0	0	0	1	0	0	0000
SUB	00000	01	1	0	0	0	0	1	0	0	0001
LW	01100	–	0	1	0	1	1	0	0	0010	
SW	01101	–	–	1	0	1	–	0	0	0	0010
BEQ	01110	–	–	0	0	0	–	0	1	0	0110
JAL	10001	–	0	0	0	0	0	1	0	1	xxxx
JR	00011	11	–	0	0	0	–	0	0	1	xxxx

2.2 Control Signals

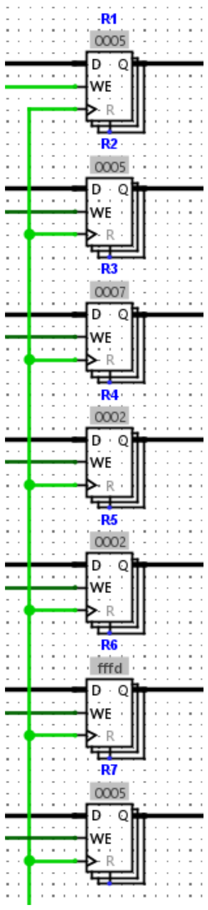
SIGNAL	SOURCE LOGIC	DESCRIPTION
RegDst	3-way priority MUX selecting one hot from: - R3_Type (all R-type non-shift/reverse) - R1_Type (shift/SLT group) - R2_Type (reverse group) - LUI, JAL	Chooses destination register index: `00`=Rt (I-type), `01`=Rd (R-type), `10`=R3 field, `11`=link reg
RegWrite	OR of its "write-back" instruction lines: `ADDI` ∨ `LW` ∨ `SW` ∨ `BEQ`/`BNE` ∨ `J` ∨ `SW` (see bottom-left OR-gate)	Enables register file write on WB stage
ALUSrc	OR-gate combining I-type data-immediate ops: `ADDI` ∨ `LW` ∨ `SW`	Selects ALU B-input source: BusB (0) vs. immediate (1)
MemRead	Directly from the `LW` decode line	When high, Data Memory drives its output onto **ReadData**
MemWrite	Directly from the `SW` decode line	When high (on clock edge), BusB is written into Data Memory at ALUResult address
MemToReg	Equals **MemRead** (wired)	Selects WB-stage MUX: ALUResult (0) vs. ReadData (1)
Branch	OR-gate of `BEQ` and `BNE` decode lines	Indicates conditional branch; used with **Zero** flag to choose PC target
Jump	OR-gate of `J`, `JAL`, and `JR` decode lines	Indicates unconditional jump (PC←target or PC←Rs)
PCSrc[2:0] (encoded)	3-bit bus from the small "PCSignals" decoder (BEQ/BNE/J/JAL/JR)	Controls the 4-way PC MUX: +1 (000), branch (001), jump-imm (010), JR (011)
ExtOp	OR-gate of `XORI`, `ORI`, and `ANDI`	When high, forces zero-extension of the 5-bit immediate
ImmSel	OR-gate of `LUI` and `JAL`	When high, selects 11-bit zero-extended immediate instead of sign-extended 5-bit
ALUOp[3:0]	16-to-1 MUX with inputs driven by each decoded R/I-type operation line (ADD, SUB, AND, OR, … REVH)	Chooses which internal ALU sub-operation to perform

3 Simulation and Testing

The Following Code Blocks are testing the whole CPU Instructions Set

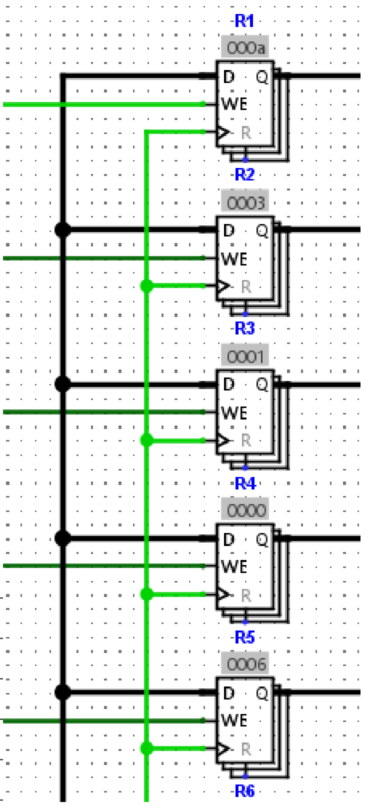


#	Assembly Code	Code(Hex)	
0	addi $1, $0, 5	4141	R1 = 0005
1	or $2, $1, $0	0A88	R2 = 0005
2	ori $3, $0, 7	51C3	R3 = 0007
3	xor $4, $3, $2	0D1A	R4 = 0002
4	and $5, $3, $4	095C	R5 = 0002
5	nor $6, $0, $4	0F84	R6 = FFFD
6	andi $7, $3, 5	495F	R7 = 0005

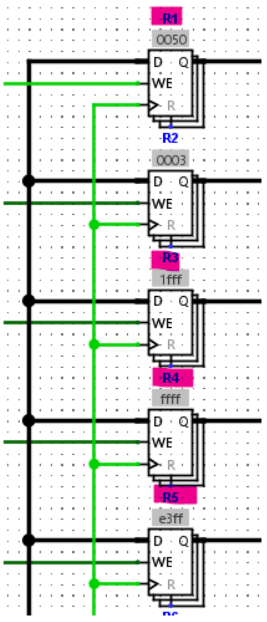


is being tested as illustrated in this example with data dependences

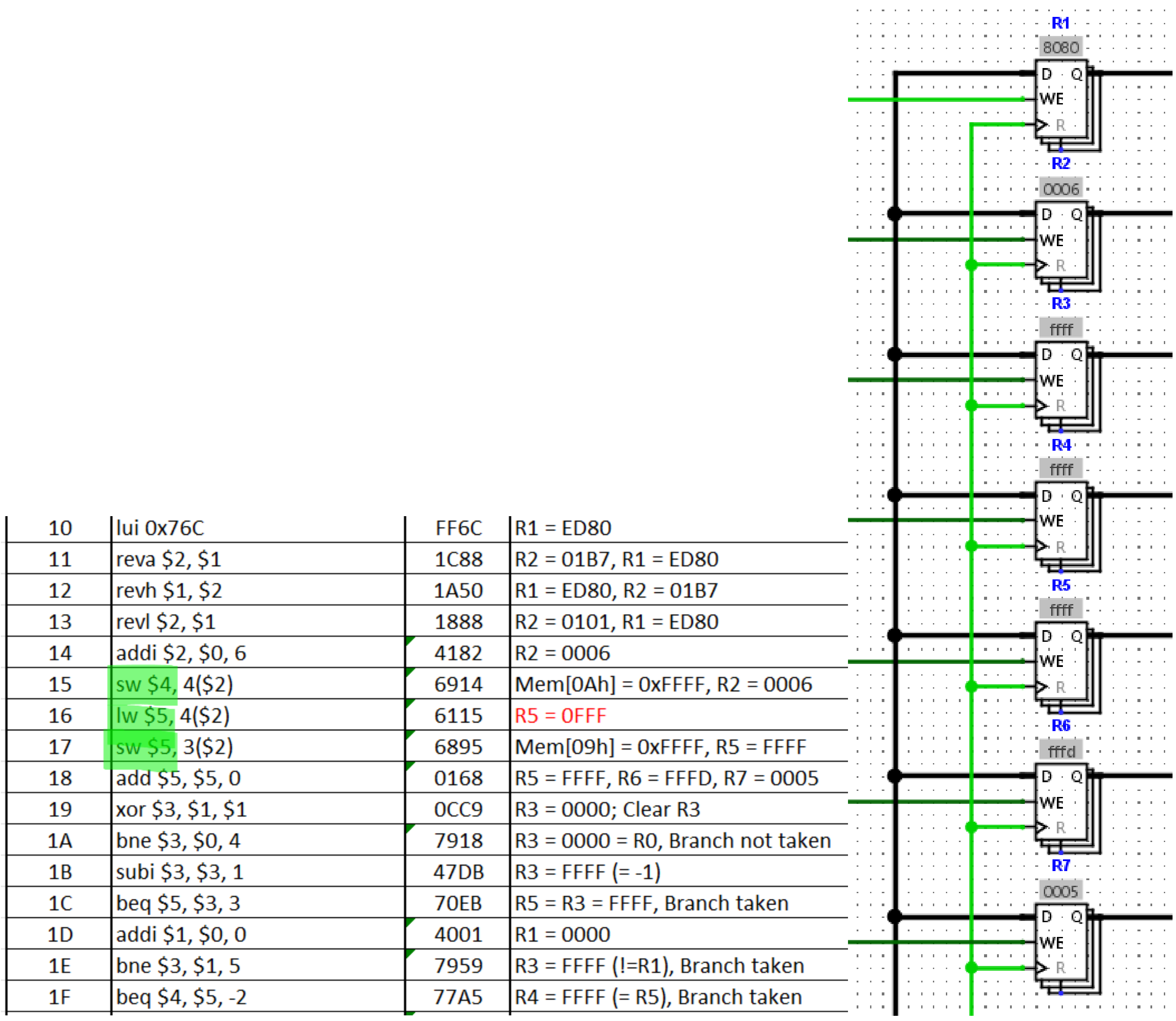
7	add $1, $1, $2	004A	R1 = 000A
8	sub $2, $1, $3	028B	R2 = 0003
9	slt $3, $6, $2	04F2	R3 = 0001
A	sltu $4, $6, $2	0732	R4 = 0000
B	xori $5, $2, 5	5955	R5 = 0006



C	sll $1, $1, $2	104a	R1 = 00A0; R1 = 0050
D	srl $3, $6, $2	12f2	R3 = 0FFF; R3 = 1FFF
E	sra $4, $6, $2	1532	R4 = FFFF
F	ror $5, $3, $2	175A	R5 = F0FF; R5 = E3FF

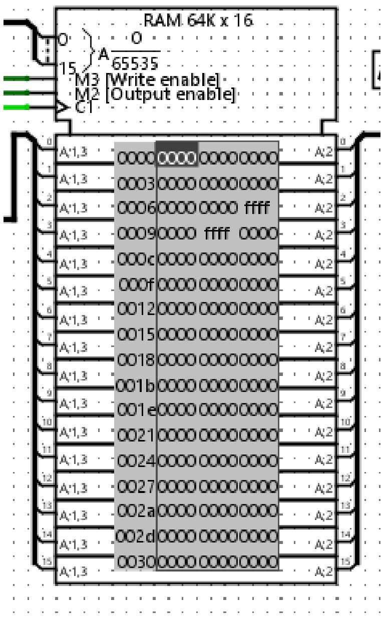


Result after reaching **0x77a5 in this test below,**



10	lui 0x76C	FF6C	R1 = ED80
11	reva $2, $1	1C88	R2 = 01B7, R1 = ED80
12	revh $1, $2	1A50	R1 = ED80, R2 = 01B7
13	revl $2, $1	1888	R2 = 0101, R1 = ED80
14	addi $2, $0, 6	4182	R2 = 0006
15	sw $4, 4($2)	6914	Mem[0Ah] = 0xFFFF, R2 = 0006
16	lw $5, 4($2)	6115	R5 = 0FFF
17	sw $5, 3($2)	6895	Mem[09h] = 0xFFFF, R5 = FFFF
18	add $5, $5, 0	0168	R5 = FFFF, R6 = FFFD, R7 = 0005
19	xor $3, $1, $1	0CC9	R3 = 0000; Clear R3
1A	bne $3, $0, 4	7918	R3 = 0000 = R0, Branch not taken
1B	subi $3, $3, 1	47DB	R3 = FFFF (= -1)
1C	beq $5, $3, 3	70EB	R5 = R3 = FFFF, Branch taken
1D	addi $1, $0, 0	4001	R1 = 0000
1E	bne $3, $1, 5	7959	R3 = FFFF (!=R1), Branch taken
1F	beq $4, $5, -2	77A5	R4 = FFFF (= R5), Branch taken

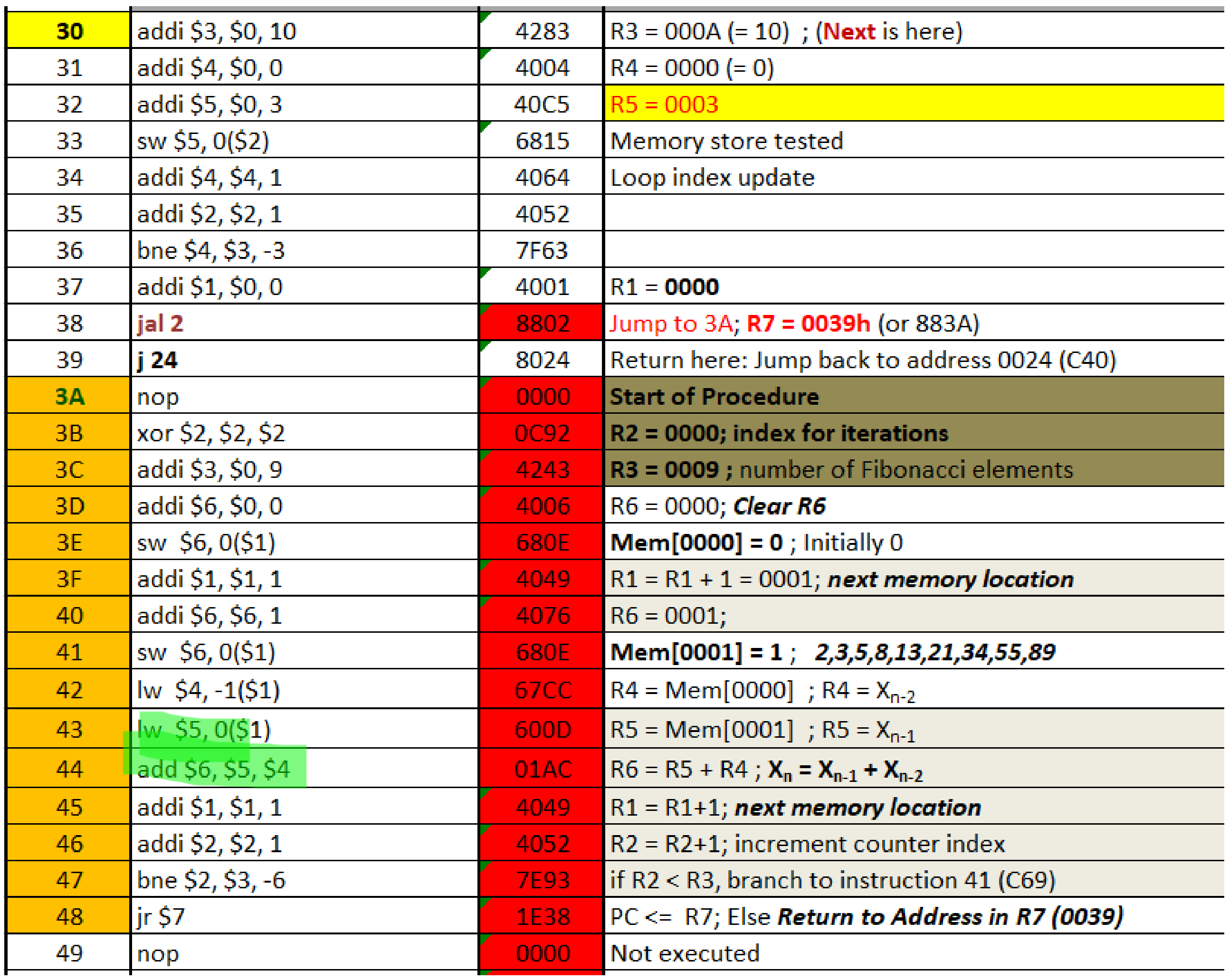
is being tested as illustrated in this example when the last instruction stores a value that has just been loaded in the instruction before it as shown:



```
0000 0000 0000 0000
0003 0000 0000 0000
0006 0000 0000 ffff
0009 0000 ffff 0000
000c 0000 0000 0000
000f 0000 0000 0000
0012 0000 0000 0000
0015 0000 0000 0000
0018 0000 0000 0000
001b 0000 0000 0000
001e 0000 0000 0000
0021 0000 0000 0000
0024 0000 0000 0000
0027 0000 0000 0000
002a 0000 0000 0000
002d 0000 0000 0000
0030 0000 0000 0000
```

Fibonacci Array

```
     addi  $3, $0, 10       # R3 ← loop count (10)
      addi  $4, $0, 0        # R4 ← index = 0
      addi  $5, $0, 3        # R5 ← value_to_store = 3

loop:   sw    $5, 0($2)        # MEM[R2] ← 3
        addi  $4, $4, 1        # index++
        addi  $2, $2, 1        # pointer++
        bne   $4, $3, loop     # if index ≠ 10, repeat

      addi  $1, $0, 0        # R1 ← output_ptr_base (start addr)
      jal   fib             # call fib(), R7 ← return addr
      j     exit            # simple halt

fib:    xor   $2, $2, $2       # R2 ← counter = 0
        addi  $3, $0, 9        # R3 ← how many values (9)
        addi  $6, $0, 0        # R6 ← Fib(0) = 0
        sw    $6, 0($1)        # MEM[R1] ← 0
        addi  $1, $1, 1        # R1++
        addi  $6, $6, 1        # R6 ← Fib(1) = 1
        sw    $6, 0($1)        # MEM[R1] ← 1

fib_loop:
        lw    $4, -1($1)       # R4 ← Fib(n−2)
        lw    $5,  0($1)       # R5 ← Fib(n−1)
        add   $6, $5, $4       # R6 ← Fib(n)=F(n−1)+F(n−2)
        addi  $1, $1, 1        # R1++ (store next)
        addi  $2, $2, 1        # R2++ (counter++)
        bne   $2, $3, fib_loop# loop while counter < 9

        jr    $7               # return to caller

exit:   nop                   # (halt)
```

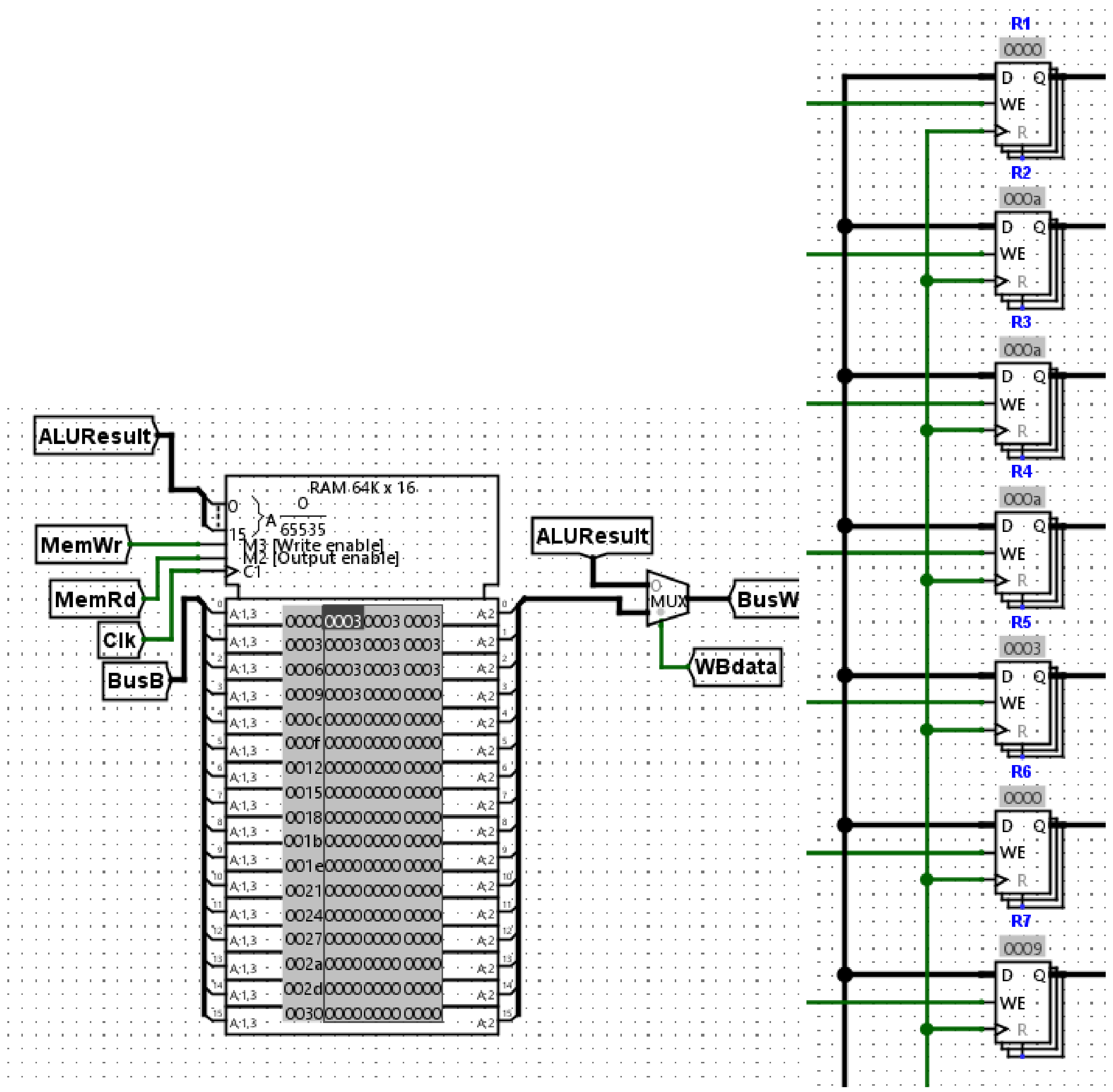


30	addi $3, $0, 10	4283	R3 = 000A (= 10) ; (**Next** is here)
31	addi $4, $0, 0	4004	R4 = 0000 (= 0)
32	addi $5, $0, 3	40C5	R5 = 0003
33	sw $5, 0($2)	6815	Memory store tested
34	addi $4, $4, 1	4064	Loop index update
35	addi $2, $2, 1	4052	
36	bne $4, $3, -3	7F63	
37	addi $1, $0, 0	4001	R1 = **0000**
38	**jal 2**	8802	Jump to 3A; **R7 = 0039h** (or 883A)
39	**j 24**	8024	Return here: Jump back to address 0024 (C40)
3A	nop	0000	**Start of Procedure**
3B	xor $2, $2, $2	0C92	**R2 = 0000; index for iterations**
3C	addi $3, $0, 9	4243	**R3 = 0009 ;** number of Fibonacci elements
3D	addi $6, $0, 0	4006	R6 = 0000; ***Clear R6***
3E	sw $6, 0($1)	680E	**Mem[0000] = 0** ; Initially 0
3F	addi $1, $1, 1	4049	R1 = R1 + 1 = 0001; ***next memory location***
40	addi $6, $6, 1	4076	R6 = 0001;
41	sw $6, 0($1)	680E	**Mem[0001] = 1** ; ***2,3,5,8,13,21,34,55,89***
42	lw $4, -1($1)	67CC	R4 = Mem[0000] ; R4 = X_{n-2}
43	lw $5, 0($1)	600D	R5 = Mem[0001] ; R5 = X_{n-1}
44	add $6, $5, $4	01AC	R6 = R5 + R4 ; $\mathbf{X_n = X_{n-1} + X_{n-2}}$
45	addi $1, $1, 1	4049	R1 = R1+1; ***next memory location***
46	addi $2, $2, 1	4052	R2 = R2+1; increment counter index
47	bne $2, $3, -6	7E93	if R2 < R3, branch to instruction 41 (C69)
48	jr $7	1E38	PC <= R7; Else ***Return to Address in R7 (0039)***
49	nop	0000	Not executed

is being tested as illustrated in this example when the add instruction read a variable that has just been loaded.

Result:

Data memory and Register file



```
0000 0003 0003 0003
0003 0003 0003 0003
0006 0003 0003 0003
0009 0003 0000 0000
000c 0000 0000 0000
000f 0000 0000 0000
0012 0000 0000 0000
0015 0000 0000 0000
0018 0000 0000 0000
001b 0000 0000 0000
001e 0000 0000 0000
0021 0000 0000 0000
0024 0000 0000 0000
0027 0000 0000 0000
002a 0000 0000 0000
002d 0000 0000 0000
0030 0000 0000 0000
```

Memory Stores and Loads

- In the first loop, ten consecutive memory writes all wrote the value 3 as intended, and you saw ten corresponding MemWrite pulses.
- In the Fibonacci routine, each SW and LW accessed the right addresses—your data memory contained the sequence 0,1,1,2,3,5,8,13,21.

Loop Control with BNE

- Both the store-loop and the Fibonacci loop iterated the exact number of times (10 and 9, respectively).
- The branch unit asserted Branch only when the index was not yet equal to the limit, then de-asserted when it should have fallen through.

Arithmetic and Data Paths

- ADDI and ADD correctly initialized and updated loop counters, pointers, and the running sum.
- The ALU's add functionality produced exactly the Fibonacci sums each cycle.

Jump and Return Mechanics

- The JAL to fib saved the return PC in $7, and the final JR used that to come back to the halt loop.

- Control signals Jump and JR fired precisely when expected.

Register File Behavior

- Registers held the right values at every stage:
  - After the store-loop, you saw R4 = 10, R2 = base+10.
  - After Fibonacci, R2 = 9, R3 = 9, and R6 = 21.

4 Conclusion

The single-cycle CPU executes the entire ISA in one clock tick, meeting project requirements for correctness across arithmetic, logic, memory, and control-flow operations. Future work could extend the design to a multi-cycle or pipelined version to improve performance.

5 References & Installing Logisim-evolution 3.9.0

- COE301 Project Specification (Term 242)
- Install Java ≥ 8 – download from https://www.java.com.
- Go to https://github.com/reds-heig/logisim-evolution/releases.
- Under v 3.9.0, download logisim-evolution-3.9.0.jar.
- Open a terminal in the download folder and run
- `java -jar logisim-evolution-3.9.0.jar`
- Logisim launches; open your .circ CPU file to simulate.