

# VLSI II - Assignment 5

Mehmet Eymen Ünay 040190218

Deniz Bashgoren 040180902

## RV32I Instruction Set

The table of all RV32I instructions and a short description in pseudocode:

Opcode	Name	Pseudocode
LUI	Load upper immediate	$rd \leftarrow imm \ll 12$
AUIPC	Add upper immediate to PC	$rd \leftarrow pc + offset \ll 12$
JAL	Jump and link	$rd \leftarrow pc + length(inst)$ $pc \leftarrow pc + offset$
JALR	Jump and link register	$rd \leftarrow pc + length(inst)$ $pc \leftarrow (rs1 + offset) \wedge -2$
BEQ	Branch equal	$if\ rs1 = rs2\ then\ pc \leftarrow pc + offset$
BNE	Branch not equal	$if\ rs1 \neq rs2\ then\ pc \leftarrow pc + offset$
BLT	Branch less than	$if\ rs1 < rs2\ then\ pc \leftarrow pc + offset$
BGE	Branch greater	$if\ rs1 \geq rs2\ then\ pc \leftarrow pc + offset$
BLTU	Branch less than unsigned	$if\ rs1 < rs2\ then\ pc \leftarrow pc + offset$
BGEU	Branch greater than unsigned	$if\ rs1 \geq rs2\ then\ pc \leftarrow pc + offset$
LB	Load byte	$rd \leftarrow s8[rs1 + offset]$
LH	Load halfword	$rd \leftarrow s16[rs1 + offset]$
LW	Load word	$rd \leftarrow s32[rs1 + offset]$
LBU	Load byte unsigned	$rd \leftarrow u8[rs1 + offset]$
LHU	Load halfword unsigned	$rd \leftarrow u16[rs1 + offset]$
SB	Store byte	$u8[rs1 + offset] \leftarrow rs2$
SH	Store halfword	$u16[rs1 + offset] \leftarrow rs2$
SW	Store word	$u32[rs1 + offset] \leftarrow rs2$
ADDI	Add immediate	$rd \leftarrow rs1 + sx(imm)$
SLTI	Set less than immediate	$rd \leftarrow sx(rs1) < sx(imm)$
SLTIU	Set less than immediate unsigned	$rd \leftarrow ux(rs1) < ux(imm)$
XORI	Exclusive or immediate	$rd \leftarrow ux(rs1) \wedge ux(imm)$
ORI	Or immediate	$rd \leftarrow ux(rs1) \mid ux(imm)$
ANDI	And immediate	$rd \leftarrow ux(rs1) \& ux(imm)$
SLLI	Shift left logic immediate	$rd \leftarrow ux(rs1) \ll ux(imm)$
SRLI	Shift right logic immediate	$rd \leftarrow ux(rs1) \gg ux(imm)$
SRAI	Shift right arithmetic immediate	$rd \leftarrow sx(rs1) \gg ux(imm)$
ADD	Add	$rd \leftarrow sx(rs1) + sx(rs2)$
SUB	Subtract	$rd \leftarrow sx(rs1) - sx(rs2)$
SLL	Shift left logical	$rd \leftarrow ux(rs1) \ll rs2$

Opcode	Name	Pseudocode
SLT	Set less than	$rd \leftarrow sx(rs1) < sx(rs2)$
SLTU	Set less than unsigned	$rd \leftarrow ux(rs1) < ux(rs2)$
XOR	Exclusive or	$rd \leftarrow ux(rs1) \wedge ux(rs2)$
SRL	Shift right logic	$rd \leftarrow ux(rs1) \gg rs2$
SRA	Shift right arithmetic	$rd \leftarrow sx(rs1) \gg rs2$
OR	Or	$rd \leftarrow ux(rs1) \mid ux(rs2)$
AND	And	$rd \leftarrow ux(rs1) \& ux(rs2)$

## Encoding of RV32I Instructions

16 of the RV32I instructions have been selected and their encoding is analyzed below.

<b>lui x30, 17</b> $x30 \leftarrow 17 \ll 12$ <b>U-TYPE</b> Fill x30's upper 20 bits with 17.	000000000000000010001 11110 0110111 ↓ ↓ ↓ imm[11:0] rd op
<b>auipc x13, 64</b> $x13 \leftarrow pc + 64 \ll 12$ <b>U-TYPE</b> Fill x13 with pc+64<<12.	0000000000000001000000 01101 0010111 ↓ ↓ ↓ imm[11:0] rd op
<b>jal x1, 60506</b> $x1 \leftarrow pc + 32$ $pc \leftarrow pc + 60506$ <b>J-TYPE</b> Save address of next instruction to x1, and jump 60506 addresses ahead.	0 1000101101 1 00001110 00001 1101111 ↓ ↓ ↓ ↓ ↓ ↓ imm[20] imm[10:1] imm[11] imm[19:12] rd op
<b>blt x10, x0, 62</b> if x10 < x0 then $pc \leftarrow pc + 62$ <b>B-TYPE</b> If the value in x10 is less than the value in x0, jump 62 addresses ahead.	0000011 00000 01010 100 11100 1100011 ↓ ↓ ↓ ↓ ↓ ↓ imm[12 10:5] rs2 rs1 fn imm[4:1 11] op
<b>srai x21, x15, 3</b> $x21 \leftarrow sx(x15) \gg 3$ <b>I-TYPE</b> Shift bits stored in x15 3 bits to the right, while filling 3 new empty bits with the leftmost bit's value, and put it inside x21.	0100000 00011 01111 101 10101 0010011 ↓ ↓ ↓ ↓ ↓ ↓ fn2 shamts rs1 fn1 rd op

<b>sb x15, 3(x10)</b> <b>u8[x10 + 3] ← x15</b>	00000000	01111	01010	000	00011	0100011
	↓	↓	↓	↓	↓	↓
<b>S-TYPE</b>	<i>imm[11:5]</i>	<i>rs2</i>	<i>rs1</i>	<i>fn</i>	<i>imm[4:0]</i>	<i>op</i>

Take the lowest byte from x15's value, and store it in the address x10+3.

<b>addi x15, x10, 16</b> <b>x15 ← x10 + 16</b>	0000000010000	01010	000	01111	0010011
	↓	↓	↓	↓	↓
<b>I-TYPE</b>	<i>imm[11:0]</i>	<i>rs1</i>	<i>fn</i>	<i>rd</i>	<i>op</i>

Add x10 with 16, put in x15.

<b>beq x15, x19, 432</b> <b>if x15 = x19 then</b> <b>pc ← pc + 432</b>	0001101	10011	01111	000	10000	1100011
	↓	↓	↓	↓	↓	↓
	<i>imm[12 10:5]</i>	<i>rs2</i>	<i>rs1</i>	<i>fn</i>	<i>imm[4:1 11]</i>	<i>op</i>
<b>B-TYPE</b>						

If the values in x15 and x19 are equal, jump 432 addresses ahead.

<b>xori x9, x9, -1</b> <b>x9 ← ux(x9) ^ -1</b>	1111111111111	01001	100	01001	0010011
	↓	↓	↓	↓	↓
<b>I-TYPE</b>	<i>imm[11:0]</i>	<i>rs1</i>	<i>fn</i>	<i>rd</i>	<i>op</i>

Perform an xor operation bit-by-bit of the value in x9 and -1 (all 1s), then put back in x9.

<b>and x15, x25, x9</b> <b>x15 ← ux(x25) &amp;</b> <b>ux(x9)</b>	00000000	01001	11001	111	01111	0110011
	↓	↓	↓	↓	↓	↓
<b>R-TYPE</b>	<i>fn2</i>	<i>rs2</i>	<i>rs1</i>	<i>fn1</i>	<i>rd</i>	<i>op</i>

Perform a bitwise and operation on x25 and x9 then store the result in x15.

<b>or x14, x10, x12</b> <b>x14 ← ux(x10)  </b> <b>ux(x12)</b>	00000000	01100	01010	110	01110	0110011
	↓	↓	↓	↓	↓	↓
<b>R-TYPE</b>	<i>fn2</i>	<i>rs2</i>	<i>rs1</i>	<i>fn1</i>	<i>rd</i>	<i>op</i>

Perform a bitwise or operation on x10 and x12 then store the result in x14.

<b>lbu x28, 1(x10)</b> <b>x28 ← u8[x10 + 1]</b>	0000000000001	01010	100	11100	0000011
	↓	↓	↓	↓	↓
<b>I-TYPE</b>	<i>imm[11:0]</i>	<i>rs1</i>	<i>fn</i>	<i>rd</i>	<i>op</i>

Get the 8-bit value in the memory at the address x10+1, then put it on the lowest byte of x28, while filling the upper 3 bytes with zeros.

**slli x15, x14, 5**  
 $x15 \leftarrow ux(x14) \ll 5$

0000000	00101	01110	001	01111	0010011
↓	↓	↓	↓	↓	↓
<i>fn2</i>	<i>shamt</i>	<i>rs1</i>	<i>fn1</i>	<i>rd</i>	<i>op</i>

**I-TYPE**

Shift bits stored in x14 5 bits to the left, while filling the 5 new empty bits with zeros (hence 'logic' shift, and put it inside x15.

**sw x16, 4(x25)**  
 $u32[x25 + 4] \leftarrow x16$

0000000	10000	11001	010	00100	0100011
↓	↓	↓	↓	↓	↓
<i>imm[11:5]</i>	<i>rs2</i>	<i>rs1</i>	<i>fn</i>	<i>imm[4:0]</i>	<i>op</i>

**S-TYPE**

Take the low 4 bytes from x16's value, put them in the 4 addresses starting from x25+4. The endianness is set by EEI during the execution, so it's not guaranteed, but in either case, loading the same bytes using LW instruction should give bytes in the same order.

**sub x15, x14, x8**  
 $x15 \leftarrow sx(x14) - sx(x8)$

0100000	01000	01110	000	01111	0110011
↓	↓	↓	↓	↓	↓
<i>fn2</i>	<i>rs2</i>	<i>rs1</i>	<i>fn1</i>	<i>rd</i>	<i>op</i>

**R-TYPE**

Calculate the value x14-x8 by treating both registers as signed, and store the result in x15.

**bne x15, x14, -26**  
 if  $x15 \neq x14$  then  
 $pc \leftarrow pc - 26$

1111111	01110	01111	001	00111	1100011
↓	↓	↓	↓	↓	↓
<i>imm[12 10:5]</i>	<i>rs2</i>	<i>rs1</i>	<i>fn</i>	<i>imm[4:1 11]</i>	<i>op</i>

**B-TYPE**

If the values in registers x15 and x14 are different, move 26 addresses backwards.

## The NOP and HINT instructions

NOP is a pseudoinstruction that does nothing, but occupy space. This is desired in cases like:

- When we want to make sure there is enough space between certain instructions
- In jump tables (such as in switch statements in C), where the distance in instructions needs to be precise
- At the end of the program, to make sure that the program doesn't trigger a trap

In practice, any instruction that doesn't change the state of the registers and the memory can be used as a NOP, however, the RISC-V specification defines a canonical encoding for the NOP: `addi x0, x0, 0` (in hexadecimal: 0x00000013).

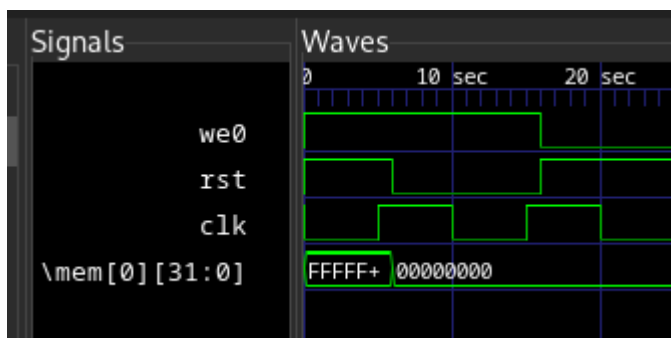
Other encodings of NOP have been named as HINT instructions in the standard. These have no effect on the visible state but may be used to communicate with the RISC-V

hardware in an implementation-specific way. For example, a hardware implementer might define that `slti x0, x1, 2` can be used to signal the hardware that the next branch is not likely to be taken. This permits the ISA users to mark branches unlikely to be taken in this way, thereby allowing the hardware to optimize its internal circuitry. The table of all reserved and non-reserved hint instructions:

Instruction	Constraints	Code Points	Purpose
LUI	$rd=x0$	$2^{20}$	<i>Reserved for future standard use</i>
AUIPC	$rd=x0$	$2^{20}$	
ADDI	$rd=x0$ , and either $rs1 \neq x0$ or $imm \neq 0$	$2^{17} - 1$	
ANDI	$rd=x0$	$2^{17}$	
ORI	$rd=x0$	$2^{17}$	
XORI	$rd=x0$	$2^{17}$	
ADD	$rd=x0$	$2^{10}$	
SUB	$rd=x0$	$2^{10}$	
AND	$rd=x0$	$2^{10}$	
OR	$rd=x0$	$2^{10}$	
XOR	$rd=x0$	$2^{10}$	
SLL	$rd=x0$	$2^{10}$	
SRL	$rd=x0$	$2^{10}$	
SRA	$rd=x0$	$2^{10}$	
FENCE	$pred=0$ or $succ=0$	$2^9 - 1$	
SLTI	$rd=x0$	$2^{17}$	<i>Reserved for custom use</i>
SLTIU	$rd=x0$	$2^{17}$	
SLLI	$rd=x0$	$2^{10}$	
SRLI	$rd=x0$	$2^{10}$	
SRAI	$rd=x0$	$2^{10}$	
SLT	$rd=x0$	$2^{10}$	
SLTU	$rd=x0$	$2^{10}$	

## Instruction Memory

In the testbench, we first load the instructions from the file `instInit.txt` into a register internal to the testbench, named `instructions`. The `instInit` file holds the instructions in hexadecimal format, however, if its binary equivalent is also provided as a separate file `instInitBin.txt`. In the testbench we first demonstrate that `rst` works asynchronously. As seen in the waveform below, the contents of the memory change to 0 right when `rst` is activated.



Next, we demonstrate that while `we=0`, the write operations won't work. First we try to fill up the memory while `we0=0`, and then while `we0=1`.

```

Address: 0 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address: 1 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address: 2 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address: 3 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address: 4 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address: 5 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address: 6 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address: 7 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
...

Address: 0 Data_Hex: 00011f37 Data_Bin: 000000000000000010001111100110111
Address: 1 Data_Hex: 00040697 Data_Bin: 00000000000000001000000011010010111
Address: 2 Data_Hex: 45b0e0ef Data_Bin: 010001011011000011100000111011111
Address: 3 Data_Hex: 06054e63 Data_Bin: 000001100000010101001110011000111
Address: 4 Data_Hex: 4037da93 Data_Bin: 010000000011011111011010100100111
Address: 5 Data_Hex: 00f501a3 Data_Bin: 000000001111010100000001101000111
Address: 6 Data_Hex: 01050793 Data_Bin: 000000010000010100000111100100111
Address: 7 Data_Hex: 1b378863 Data_Bin: 000110110011011110001000011000111
...

```

Running the testbench repeatedly emits the warning: VCD warning: array word InstMem\_Tb.m.mem[0] will conflict with an escaped identifier. . It can be suppressed as explained in <https://stackoverflow.com/questions/20317820/>. We decided to keep the code simpler at the cost of this warning.

## Data Memory

wr\_strb is encoded as following:

```

input wire [2:0] wr_strb;
// wr_strb = 000 -> store word
// wr_strb = 001 -> store lower halfword
// wr_strb = 010 -> nop
// wr_strb = 011 -> store higher halfword
// wr_strb = 100 -> store lowest byte
// wr_strb = 101 -> store 2nd lowest byte
// wr_strb = 110 -> store 3rd lowest byte
// wr_strb = 111 -> store highest byte

```

The state 010 is chosen to be nop to simplify the hardware: the msb bit controls the byte-level stores; the lsb bit controls the halfword-level stores and the combination 000 is for word-level stores. The enum-like encoding of wr\_strb allows us increment it in the testbench as if it was an integer. In the testbench, a value of 0xAABBCCDD is placed in the addresses starting from 0, but each time with a new wr\_strb mode. The values are stored in little endian order. In the testbench we've redefined DEPTH=8 so that the console output is more consice, however it can easily be changed to its original value of 128. The end result is as following:

```

...
Address: 0 Data_Hex: aabbccdd Data_Bin: 10101010101110111100110011011101
Address: 1 Data_Hex: 0000ccdd Data_Bin: 00000000000000001100110011011101
Address: 2 Data_Hex: 00000000 Data_Bin: 00000000000000000000000000000000
Address: 3 Data_Hex: ccdd0000 Data_Bin: 11001100110111010000000000000000
Address: 4 Data_Hex: 000000dd Data_Bin: 00000000000000000000000011011101
Address: 5 Data_Hex: 0000dd00 Data_Bin: 00000000000000001101110100000000
Address: 6 Data_Hex: 00dd0000 Data_Bin: 00000000110111010000000000000000
Address: 7 Data_Hex: dd000000 Data_Bin: 11011101000000000000000000000000

```

# Regfile

The full OpenLane flow is performed. According to reports/signoff/25-rcx\_sta.rpt file:

## No Violations

25-rcx\_sta.rpt file shows there is no max slew or max cap violation :

=====

report\_check\_types -max\_slew -max\_cap -max\_fanout -violators

=====

max fanout

Pin	Limit	Fanout	Slack
-----			
_5957_/X	10	24	-14 (VIOLATED)
input40/X	10	24	-14 (VIOLATED)
_5167_/X	10	19	-9 (VIOLATED)
_4690_/X	10	17	-7 (VIOLATED)
_5627_/X	10	17	-7 (VIOLATED)
_5681_/X	10	17	-7 (VIOLATED)
_5762_/X	10	17	-7 (VIOLATED)
_4466_/X	10	16	-6 (VIOLATED)
clkbuf_0_clk/X	10	16	-6 (VIOLATED)
_4671_/X	10	15	-5 (VIOLATED)
_5396_/X	10	15	-5 (VIOLATED)
clkbuf_leaf_61_clk/X	10	15	-5 (VIOLATED)
_4669_/X	10	14	-4 (VIOLATED)
_4723_/X	10	14	-4 (VIOLATED)
clkbuf_leaf_114_clk/X	10	14	-4 (VIOLATED)
clkbuf_leaf_22_clk/X	10	14	-4 (VIOLATED)
clkbuf_leaf_5_clk/X	10	14	-4 (VIOLATED)
_4558_/X	10	13	-3 (VIOLATED)
_4654_/X	10	13	-3 (VIOLATED)
_4672_/X	10	13	-3 (VIOLATED)
_4708_/X	10	13	-3 (VIOLATED)
_4750_/X	10	13	-3 (VIOLATED)
_4926_/X	10	13	-3 (VIOLATED)
_5788_/X	10	13	-3 (VIOLATED)
_6277_/X	10	13	-3 (VIOLATED)
clkbuf_leaf_31_clk/X	10	13	-3 (VIOLATED)
clkbuf_leaf_39_clk/X	10	13	-3 (VIOLATED)
clkbuf_leaf_45_clk/X	10	13	-3 (VIOLATED)
clkbuf_leaf_49_clk/X	10	13	-3 (VIOLATED)
_4402_/X	10	12	-2 (VIOLATED)
_4704_/X	10	12	-2 (VIOLATED)
_4707_/X	10	12	-2 (VIOLATED)
_4756_/X	10	12	-2 (VIOLATED)
_4759_/X	10	12	-2 (VIOLATED)
_4792_/X	10	12	-2 (VIOLATED)
_5251_/X	10	12	-2 (VIOLATED)
_6221_/X	10	12	-2 (VIOLATED)
_6445_/X	10	12	-2 (VIOLATED)

clkbuf_leaf_11_clk/X	10	12	-2 (VIOLATED)
clkbuf_leaf_17_clk/X	10	12	-2 (VIOLATED)
clkbuf_leaf_93_clk/X	10	12	-2 (VIOLATED)
_4326_/X	10	11	(VIOLATED)
_4329_/X	10	11	(VIOLATED)
_4364_/X	10	11	(VIOLATED)
_4371_/X	10	11	(VIOLATED)
_4380_/X	10	11	(VIOLATED)
_4392_/X	10	11	(VIOLATED)
_4408_/X	10	11	(VIOLATED)
_4420_/X	10	11	(VIOLATED)
_4436_/X	10	11	(VIOLATED)
_4565_/X	10	11	(VIOLATED)
_4591_/X	10	11	(VIOLATED)
_4606_/X	10	11	(VIOLATED)
_4652_/X	10	11	(VIOLATED)
_4659_/X	10	11	(VIOLATED)
_4712_/X	10	11	(VIOLATED)
_4717_/X	10	11	(VIOLATED)
_4747_/X	10	11	(VIOLATED)
_4748_/X	10	11	(VIOLATED)
_4752_/X	10	11	(VIOLATED)
_4853_/X	10	11	(VIOLATED)
_4881_/X	10	11	(VIOLATED)
_4996_/X	10	11	(VIOLATED)
_5185_/X	10	11	(VIOLATED)
_5252_/X	10	11	(VIOLATED)
_5266_/X	10	11	(VIOLATED)
_5307_/X	10	11	(VIOLATED)
_5310_/X	10	11	(VIOLATED)
_6095_/X	10	11	(VIOLATED)
_6517_/X	10	11	(VIOLATED)
_6843_/X	10	11	(VIOLATED)
_7236_/X	10	11	(VIOLATED)
_8516_/X	10	11	(VIOLATED)
clkbuf_leaf_101_clk/X	10	11	(VIOLATED)
clkbuf_leaf_18_clk/X	10	11	(VIOLATED)
clkbuf_leaf_34_clk/X	10	11	(VIOLATED)
clkbuf_leaf_37_clk/X	10	11	(VIOLATED)
clkbuf_leaf_56_clk/X	10	11	(VIOLATED)
clkbuf_leaf_81_clk/X	10	11	(VIOLATED)
clkbuf_leaf_87_clk/X	10	11	(VIOLATED)
clkbuf_leaf_94_clk/X	10	11	(VIOLATED)

=====

max slew violation count 0

max fanout violation count 81

max cap violation count 0

=====

## Results



- Design area 82279 u^2 9% utilization. According to worst\_slack reports:

```
=====
report_worst_slack -max (Setup)
=====
worst slack 1.75

=====
report_worst_slack -min (Hold)
=====
worst slack 0.30
```

As the worst slack from hold can be 0.30 the minimum clock period can be 10.30 ns which makes 97MHz the maximum clock frequency.

```
=====
report_checks -unconstrained
=====
Startpoint: rst (input port clocked by clk)
Endpoint: _9279_ (recovery check against rising-edge clock clk)
Path Group: **async_default**
Path Type: max
```

Fanout	Cap	Slew	Delay	Time	Description
-----					
			0.00	0.00	clock clk (rise edge)
			0.00	0.00	clock network delay (propagated)
			2.00	2.00	^ input external delay
		0.03	0.02	2.02	^ rst (in)
1	0.01				rst (net)
		0.03	0.00	2.02	^ input11/A (sky130_fd_sc_hd__buf_6)
		0.22	0.21	2.23	^ input11/X (sky130_fd_sc_hd__buf_6)
4	0.11				net11 (net)
		0.23	0.04	2.27	^ fanout246/A (sky130_fd_sc_hd__clkbuf_4)
		0.24	0.35	2.62	^ fanout246/X (sky130_fd_sc_hd__clkbuf_4)
5	0.08				net246 (net)
		0.25	0.01	2.64	^ fanout193/A (sky130_fd_sc_hd__buf_2)
		0.22	0.31	2.95	^ fanout193/X (sky130_fd_sc_hd__buf_2)
5	0.05				net193 (net)
		0.22	0.00	2.95	^ fanout178/A (sky130_fd_sc_hd__buf_2)
		0.25	0.33	3.28	^ fanout178/X (sky130_fd_sc_hd__buf_2)
7	0.05				net178 (net)
		0.25	0.00	3.28	^ fanout177/A (sky130_fd_sc_hd__clkbuf_4)
		0.18	0.32	3.60	^ fanout177/X (sky130_fd_sc_hd__clkbuf_4)
10	0.06				net177 (net)
		0.18	0.00	3.60	^ fanout176/A (sky130_fd_sc_hd__clkbuf_4)
		0.17	0.29	3.89	^ fanout176/X (sky130_fd_sc_hd__clkbuf_4)
10	0.05				net176 (net)
		0.17	0.00	3.89	^ _9279_/RESET_B (sky130_fd_sc_hd__dfrtp_1)
				3.89	data arrival time
			10.00	10.00	clock clk (rise edge)

			0.00	10.00	clock source latency
		0.32	0.22	10.22	^ clk (in)
2	0.07				clk (net)
		0.32	0.00	10.22	^ clkbuf_0_clk/A (sky130_fd_sc_hd__clkbuf_16)
		0.28	0.36	10.58	^ clkbuf_0_clk/X (sky130_fd_sc_hd__clkbuf_16)
16	0.27				clknet_0_clk (net)
		0.28	0.01	10.59	^ clkbuf_4_8_0_clk/A
(sky130_fd_sc_hd__clkbuf_8)					
		0.19	0.29	10.88	^ clkbuf_4_8_0_clk/X
(sky130_fd_sc_hd__clkbuf_8)					
8	0.11				clknet_4_8_0_clk (net)
		0.19	0.01	10.89	^ clkbuf_leaf_16_clk/A
(sky130_fd_sc_hd__clkbuf_16)					
		0.05	0.17	11.06	^ clkbuf_leaf_16_clk/X
(sky130_fd_sc_hd__clkbuf_16)					
8	0.03				clknet_leaf_16_clk (net)
		0.05	0.00	11.06	^ _9279_/CLK (sky130_fd_sc_hd__dfrrtp_1)
			-0.25	10.81	clock uncertainty
			0.00	10.81	clock reconvergence pessimism
			0.20	11.01	library recovery time
				11.01	data required time
-----					
				11.01	data required time
				-3.89	data arrival time
-----					
				7.13	slack (MET)

Startpoint: rd\_addr0[2] (input port clocked by clk)

Endpoint: rd\_dout0[10] (output port clocked by clk)

Path Group: clk

Path Type: max

Fanout	Cap	Slew	Delay	Time	Description
-----					
			0.00	0.00	clock clk (rise edge)
			0.00	0.00	clock network delay (propagated)
			2.00	2.00	^ input external delay
		0.04	0.02	2.02	^ rd_addr0[2] (in)
1	0.01				rd_addr0[2] (net)
		0.04	0.00	2.02	^ input3/A (sky130_fd_sc_hd__buf_6)
		0.25	0.24	2.26	^ input3/X (sky130_fd_sc_hd__buf_6)
9	0.12				net3 (net)
		0.25	0.02	2.29	^ _4854_/A (sky130_fd_sc_hd__buf_2)
		0.23	0.32	2.60	^ _4854_/X (sky130_fd_sc_hd__buf_2)
10	0.05				_1516_ (net)
		0.23	0.00	2.60	^ _4856_/A_N (sky130_fd_sc_hd__and2b_1)
		0.15	0.33	2.94	v _4856_/X (sky130_fd_sc_hd__and2b_1)
6	0.03				_1518_ (net)
		0.15	0.00	2.94	v _4936_/A (sky130_fd_sc_hd__and2_1)
		0.14	0.29	3.23	v _4936_/X (sky130_fd_sc_hd__and2_1)
4	0.03				_1597_ (net)

```

0.14 0.00 3.23 v _4937_/A (sky130_fd_sc_hd__clkbuf_4)
0.15 0.29 3.52 v _4937_/X (sky130_fd_sc_hd__clkbuf_4)
10 0.07 _1598_ (net)
0.15 0.01 3.53 v _5241_/A2 (sky130_fd_sc_hd__a2111o_1)
0.08 0.48 4.01 v _5241_/X (sky130_fd_sc_hd__a2111o_1)
1 0.01 _1893_ (net)
0.08 0.00 4.01 v _5255_/B (sky130_fd_sc_hd__or4_1)
0.11 0.57 4.58 v _5255_/X (sky130_fd_sc_hd__or4_1)
1 0.01 _1907_ (net)
0.11 0.00 4.58 v _5269_/A (sky130_fd_sc_hd__or4_2)
0.21 0.89 5.47 v _5269_/X (sky130_fd_sc_hd__or4_2)
1 0.04 _1921_ (net)
0.21 0.00 5.47 v _5270_/A (sky130_fd_sc_hd__buf_4)
0.10 0.29 5.76 v _5270_/X (sky130_fd_sc_hd__buf_4)
1 0.08 net51 (net)
0.11 0.02 5.78 v output51/A (sky130_fd_sc_hd__buf_2)
0.09 0.22 6.00 v output51/X (sky130_fd_sc_hd__buf_2)
1 0.03 rd_dout0[10] (net)
0.09 0.00 6.00 v rd_dout0[10] (out)
6.00 data arrival time

10.00 10.00 clock clk (rise edge)
0.00 10.00 clock network delay (propagated)
-0.25 9.75 clock uncertainty
0.00 9.75 clock reconvergence pessimism
-2.00 7.75 output external delay
7.75 data required time

-----
7.75 data required time
-6.00 data arrival time
-----

1.75 slack (MET)

=====
report_checks --slack_max -0.01
=====
No paths found.
```

The critical path is caused by the reading of the memory. Reset takes less time, almost half of reading the memory. The reason for the critical path is high fan-in. Despite the synthesizer made use of buffers and used smaller input sized logic gates, to read 32 registers and direct it to the output requires high overall fan-in.