512-Row SRAM Decoder Design and Verification in TSMC N16ADFP

Nandini Kumawat

November 12, 2025

Guided by: Prof. Chris Kim

UNIVERSITY OF MINNESOTA
Driven to Discover®

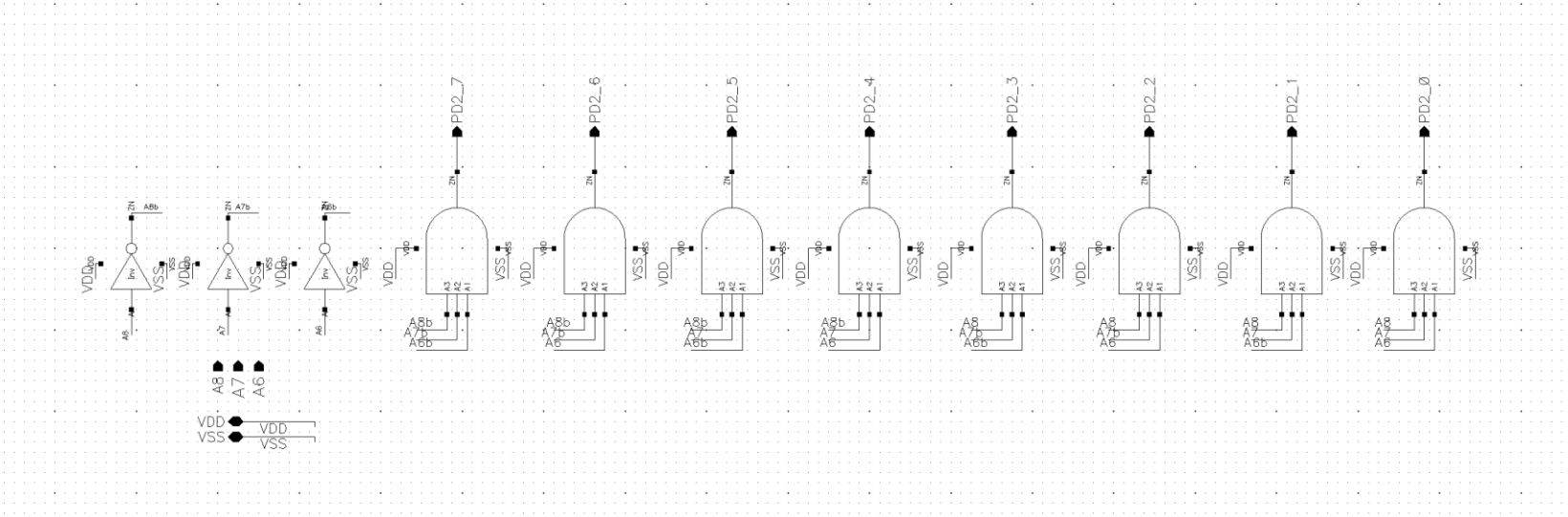
Introduction

This report presents the design and verification of a 512-row SRAM row decoder implemented in TSMC 16nm (N16ADFP) CMOS technology. The decoder is responsible for selecting one of 512 wordlines based on a 9-bit address input. The entire design process was automated using Cadence SKILL scripting, enabling **full automation** of schematic generation and verification.

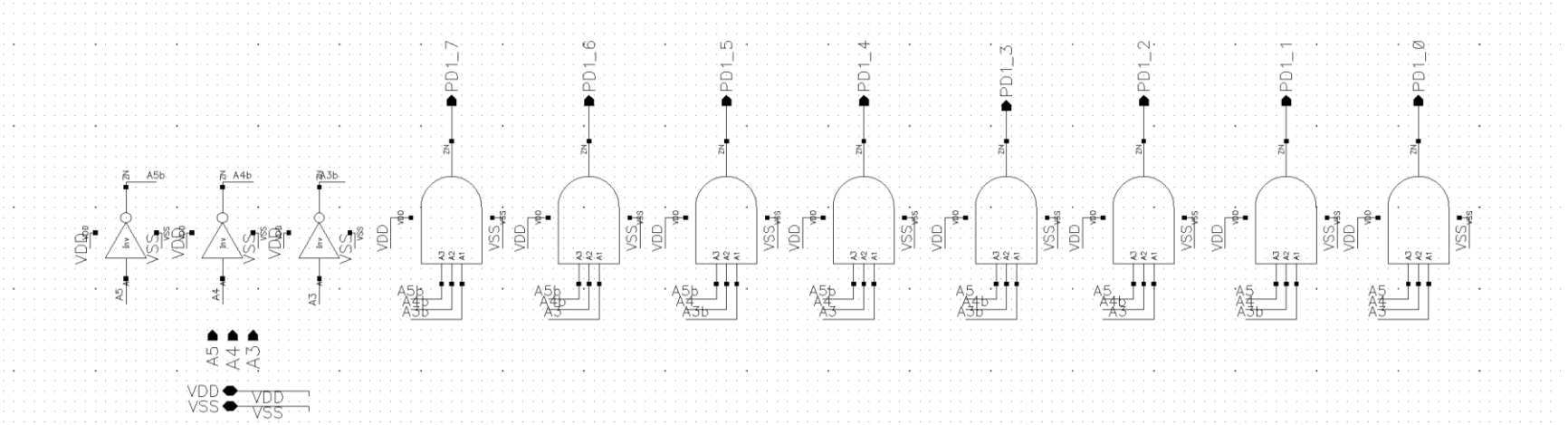
Decoder + Pre-Decoder Architecture

The 512-row decoder is driven by a 9-bit address (A8:A0) plus a clock (CLK) signal used for gating. Instead, the design uses a **three-level hierarchical decoder** structure, breaking the 9-bit address into smaller groups and decoding them in stages. The address bits are partitioned as follows:

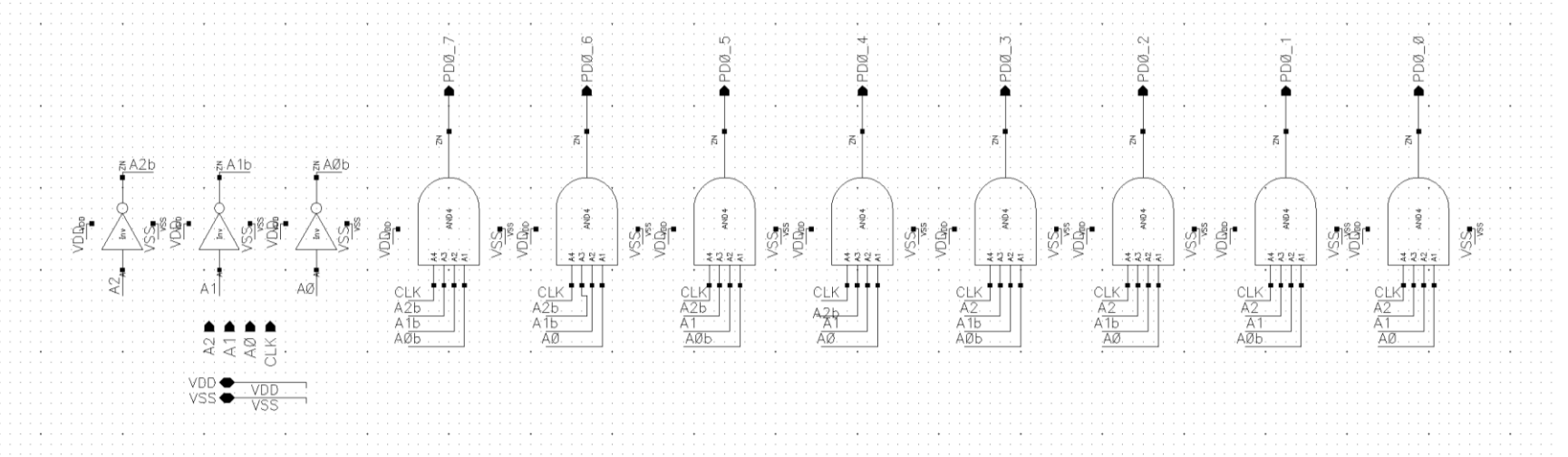
- **PD2**: Bits A8:A6 (3 bits) – *Block select*



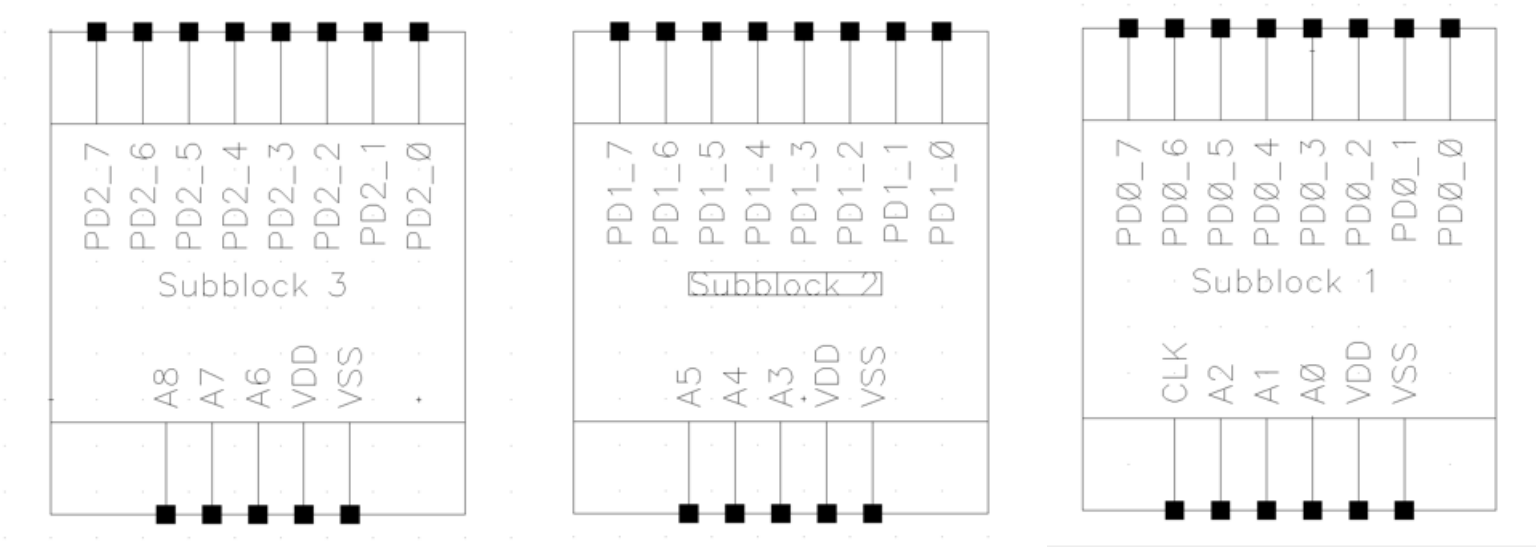
- **PD1**: Bits A5:A3 (3 bits) – *Group select*



- **PD0**: Bits A2:A0 (3 bits, plus clock) – *Row select (with clock gating)*



Each of these is a predecoder block that generates 8 outputs (since 3 address bits have 2^3 = 8 combinations). In effect, the single 9-to-512 decoding logic is factorized into three smaller decoders whose outputs are then combined. PD2 selects one *block* of rows (out of 8 blocks of 64 rows each), PD1 selects one *group* of 8 rows within that block, and PD0 selects one *row* within the group. The final decoding stage consists of an array of 512 three-input AND gates. Each AND gate takes one output from PD2, one from PD1, and one from PD0, producing a unique wordline WL<i> for *i* from 0 to 511. Thus, **each wordline is asserted only when the correct combination of PD0, PD1, and PD2 outputs are all active**.



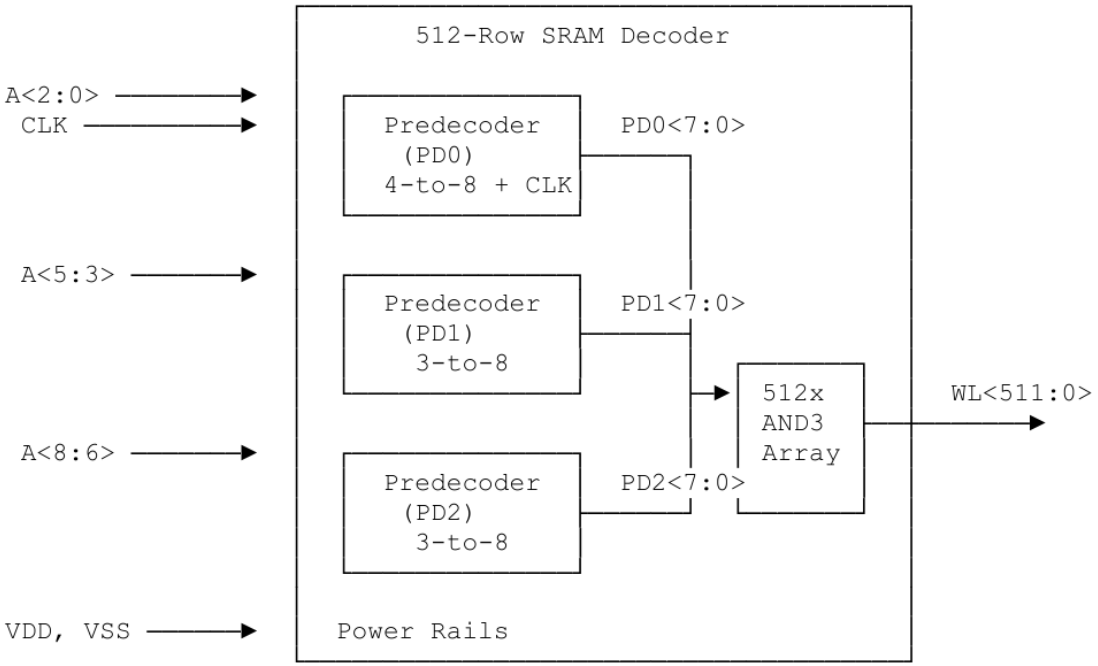
(Figure 1: High-level block diagram of the 512-row hierarchical predecoder, showing the PD2, PD1, and PD0 predecoder blocks feeding the 3-input AND matrix)

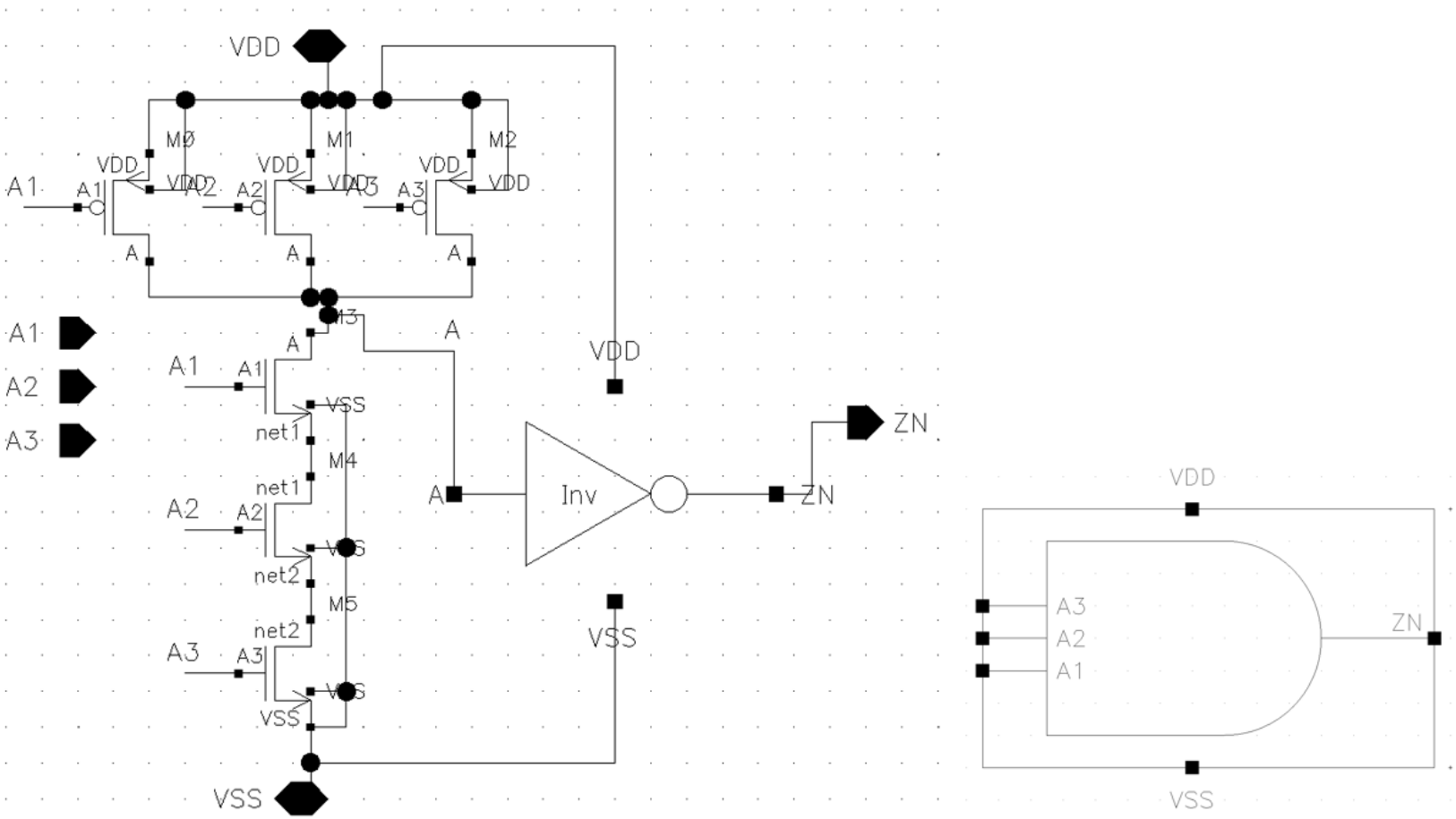
AND3 Matrix Structure

The final stage of decoding is a matrix of 512 AND gates (3-input). Each wordline **WL<i>** is generated by taking one output from each predecoder:

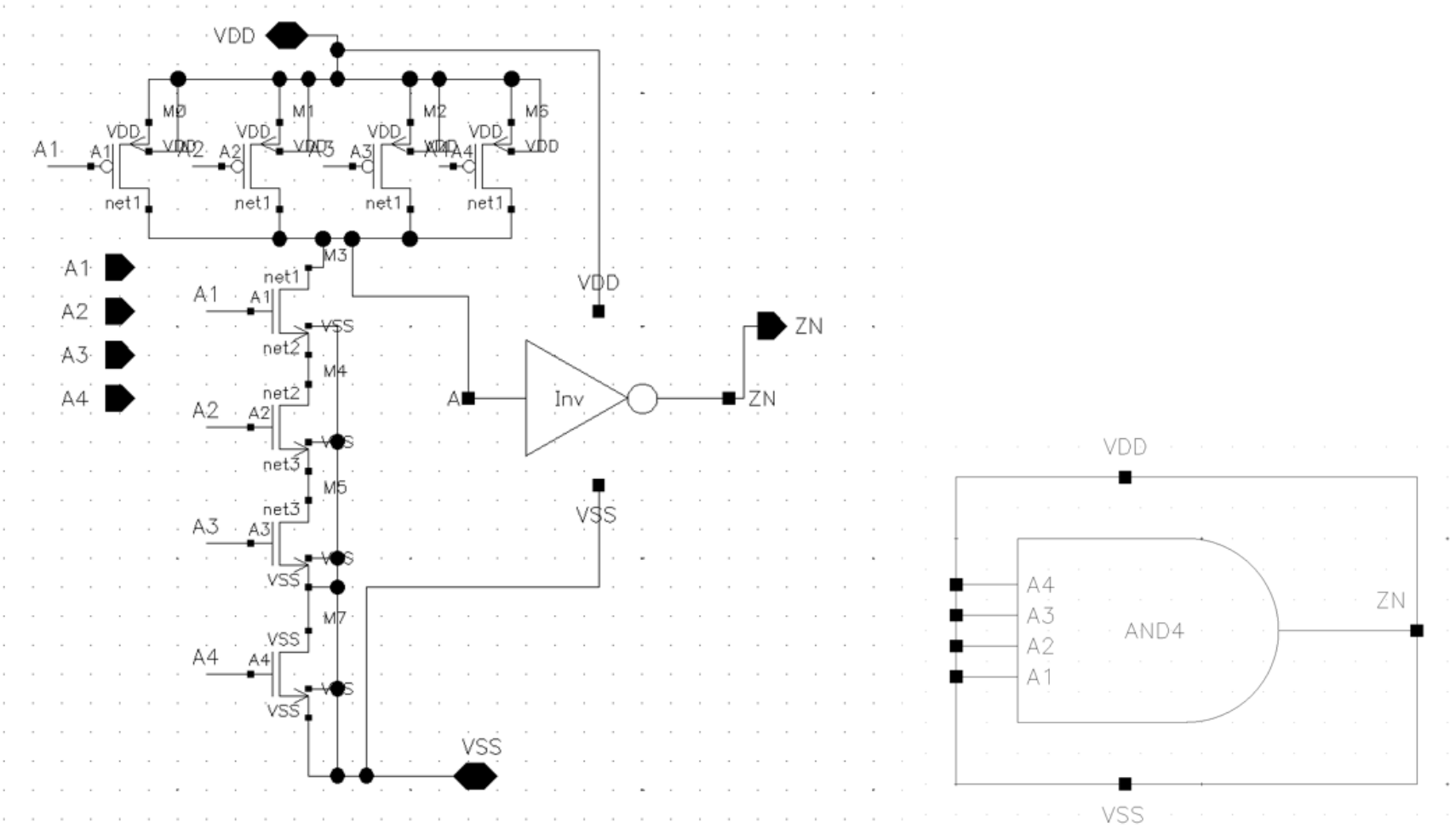
$$\mathrm{WL} < i >= \mathrm{PD2}[x] \ \wedge \ \mathrm{PD1}[y] \ \wedge \ \mathrm{PD0}[z],$$

All 512 such gates are placed in the top-level decoder schematic, effectively forming an 8×8×8 combinational matrix. To keep the schematic organized, the instances were algorithmically named and placed in a grid pattern via the SKILL script (though the final visual placement was adjusted as needed).





Transistor-level schematics of the standard cells used: the INV (inverter/buffer), the 3-input NAND (ND3), and the 4-input NAND (ND4).



Transistor-level schematics of the standard cells used: the INV (inverter/buffer), the 3-input NAND (ND3), and the 4-input NAND (ND4).

SKILL-Based Automation

Cadence SKILL was used extensively to automate both the construction and verification of the decoder. Four main SKILL scripts were developed for this project, each targeting a specific aspect of the design workflow:

- **sram_decoder_512_hierarchical.il – Schematic Generator:** This script programmatically builds the entire decoder schematic ("correct-by-construction"). It creates the library and cell views, then instantiates the three predecoder subcells (PD0, PD1, PD2) and 512 copies of the 3-input AND gate cell. It also creates and names nets for all interconnections: the address inputs (A0:8), the predecoder outputs (pd0_0…pd0_7, etc.), the wordlines (WL<0>…WL<511>), as well as power nets (VDD, VSS) and the clock. The script connects each AND instance's inputs to the appropriate PD0/1/2 outputs according to the decoding logic. Instance naming and net naming conventions were carefully designed (e.g., instances I3 through I514 corresponding to each AND gate, and instance I0/I1/I2 for PD0/PD1/PD2) to reflect the hierarchy and facilitate verification. By automating schematic construction, the design avoids wiring mistakes and ensures that **all 512 wordlines and 515 instances** are created and connected exactly as intended, with consistent naming.

```
; sram_decoder_512_hierarchical.il
; Purpose: 512-row SRAM decoder using hierarchical predecoder blocks
; Library: ProjectSRAM
; Blocks: subblock1 (PD0), subblock2 (PD1), subblock3 (PD2), and3
; Top pins: VDD, VSS, CLK, A<8:0>, WL<511:0>
;
; Usage (CIW):
;   setWorkingDir("/home/kumaw010/tsmcN16_Nandini")
;   load("sram_decoder_512_hierarchical.il")
;   buildDecoder512Hierarchical("SRAM_DCDR" "rowDecoder512")
;   deOpenCellView("SRAM_DCDR" "rowDecoder512" "schematic")

; ---------------------------------
; Helpers
```

```
; ---------------------------------
procedure( sdEnsureLib(lib tech)
  let( (libObj)
    libObj = ddGetObj(lib)
    unless(libObj
      libObj = ddCreateLib(lib)
      when(tech && tech != nil
        printf("Note: Manually attach tech library '%s' to library '%s' if needed\n" tech
lib)
      )
    )
    libObj
  )
)

procedure( sdOpen(cvLib cvCell cvView)
  let( (cv)
    cv = dbOpenCellViewByType(cvLib cvCell cvView "" "a")
    unless(cv cv = dbOpenCellViewByType(cvLib cvCell cvView cvView "a"))
    cv
  )
)

procedure( sdNet(cv name)
  let( (n)
    n = dbFindNetByName(cv name)
    unless(n n = dbCreateNet(cv name))
    n
  )
)

procedure( sdPin(cv name dir)
  let( (termObj net)
    termObj = dbFindTermByName(cv name)
```

```
    unless(termObj
      net = sdNet(cv name)
      termObj = dbCreateTerm(net name dir)
    )
    termObj
  )
)

procedure( sdInst(cv lib cell view xy orient)
  let( (useView cvTmp inst xPos yPos)
    ; Extract x and y from list
    xPos = car(xy)
    yPos = cadr(xy)

    useView = view
    cvTmp = dbOpenCellViewByType(lib cell useView "" "r")

    unless(cvTmp
      useView = "symbol"
      cvTmp = dbOpenCellViewByType(lib cell useView "" "r")
    )

    unless(cvTmp
      error("Cannot find cellview %s/%s/%s or symbol" lib cell view)
    )

    ; Use schCreateInst instead of dbCreateInst - this properly creates instTerms!
    inst = schCreateInst(cv cvTmp nil xPos:yPos orient)

    dbClose(cvTmp)

    inst
  )
)
```

```
procedure( sdConnectPin(inst pinName netName cv)
  let( (net)
    net = sdNet(cv netName)

    ; Loop through instance terminals and connect
    foreach(term inst~>instTerms
      when(term~>name == pinName
        term~>net = net
      )
    )
  )
)

; ---------------------------------
; Instantiate Predecoder Blocks
; ---------------------------------
procedure( sdInstPredecoder(cv lib cell x y netMap)
  let( (inst pinName netName)
    inst = sdInst(cv lib cell "symbol" list(x y) "R0")

    ; Connect all pins based on netMap
    foreach(pair netMap
      pinName = car(pair)
      netName = cadr(pair)
      sdConnectPin(inst pinName netName cv)
    )

    inst
  )
)

; ---------------------------------
```

```
; Instantiate 512 AND3 gates for WL decode
; --------------------------------
procedure( sdRowDecode512(cv lib and3Cell x y dx dy)
  let( (i j k idx inst pd0Net pd1Net pd2Net wlNet)
    for(i 0 7
      for(j 0 7
        for(k 0 7
          idx = i*64 + j*8 + k

          ; Create AND3 instance
          inst = sdInst(cv lib and3Cell "symbol"
                        list(x + i*dx, y + j*dy + k*3) "R0")

          ; Connect inputs
       pd0Net = sprintf(nil "pd0_%d" i)
       pd1Net = sprintf(nil "pd1_%d" j)
       pd2Net = sprintf(nil "pd2_%d" k)
          wlNet  = sprintf(nil "WL<%d>" idx)

          sdConnectPin(inst "A1" pd0Net cv)
          sdConnectPin(inst "A2" pd1Net cv)
          sdConnectPin(inst "A3" pd2Net cv)
          sdConnectPin(inst "ZN" wlNet cv)
          sdConnectPin(inst "VDD" "VDD" cv)
          sdConnectPin(inst "VSS" "VSS" cv)
        )
      )
    )
  )
)

; --------------------------------
; Top builder
; --------------------------------
```

```
procedure( buildDecoder512Hierarchical(libName topCell)
  let( (cv i pd0Map pd1Map pd2Map)
    unless(libName libName = "SRAM_DCDR")
    unless(topCell topCell = "rowDecoder512")

    sdEnsureLib(libName nil)
    cv = sdOpen(libName topCell "schematic")

    ; Create top-level pins
    sdPin(cv "VDD" "input")
    sdPin(cv "VSS" "input")
    sdPin(cv "CLK" "input")
    for(i 8 0 -1 sdPin(cv sprintf(nil "A<%d>" i) "input"))
    sdPin(cv "WL<511:0>" "output")

    ; Build pin maps for predecoder blocks
    ; PD0: A0, A1, A2, CLK -> PD0_0..PD0_7
    pd0Map = list(
      list("A0" "A<0>")
      list("A1" "A<1>")
      list("A2" "A<2>")
      list("CLK" "CLK")
      list("VDD" "VDD")
      list("VSS" "VSS")
    )
    for(i 0 7
      pd0Map = append(pd0Map list(list(sprintf(nil "PD0_%d" i) sprintf(nil "PD0_%d" i))))
    )

    ; PD1: A3, A4, A5 -> PD1_0..PD1_7
    pd1Map = list(
      list("A3" "A<3>")
      list("A4" "A<4>")
      list("A5" "A<5>")
```

```
      list("VDD" "VDD")
      list("VSS" "VSS")
    )
    for(i 0 7
      pd1Map = append(pd1Map list(list(sprintf(nil "PD1_%d" i) sprintf(nil "PD1_%d" i))))
    )

    ; PD2: A6, A7, A8 -> PD2_0..PD2_7
    pd2Map = list(
      list("A6" "A<6>")
      list("A7" "A<7>")
      list("A8" "A<8>")
      list("VDD" "VDD")
      list("VSS" "VSS")
    )
    for(i 0 7
      pd2Map = append(pd2Map list(list(sprintf(nil "PD2_%d" i) sprintf(nil "PD2_%d" i))))
    )

    ; Instantiate the three predecoder blocks
    printf("Instantiating subblock1 (PD0)...\n")
    sdInstPredecoder(cv "ProjectSRAM" "subblock1" 0 0 pd0Map)

    printf("Instantiating subblock2 (PD1)...\n")
    sdInstPredecoder(cv "ProjectSRAM" "subblock2" 50 0 pd1Map)

    printf("Instantiating subblock3 (PD2)...\n")
    sdInstPredecoder(cv "ProjectSRAM" "subblock3" 100 0 pd2Map)

    ; Instantiate 512 AND3 gates for row decode
    printf("Instantiating 512 AND3 gates for WL decode...\n")
    sdRowDecode512(cv "ProjectSRAM" "and3" 150 0 20 30)

; Reconnect AND3 outputs to WL nets
```

```
printf("Reconnecting AND3 outputs to WL nets...\n")
for(i 0 7
  for(j 0 7
    for(k 0 7
      let( (idx wlName wlNet andInst)
        idx = i*64 + j*8 + k
        wlName = sprintf(nil "WL<%d>" idx)

        ; Create WL net
        wlNet = dbFindNetByName(cv wlName)
        unless(wlNet
          wlNet = dbCreateNet(cv wlName)
        )

        ; Find the AND3 instance and reconnect output
        foreach(inst cv~>instances
          when(inst~>cellName == "and3"
            foreach(term inst~>instTerms
              when(lowerCase(term~>name) == "zn"
                ; Check if this is connected to the right pd nets
                let( (a1Net a2Net a3Net pd0Net pd1Net pd2Net)
                  foreach(iterm inst~>instTerms
                    case(lowerCase(iterm~>name)
                   ("a1" when(iterm~>net a1Net = iterm~>net~>name))
                   ("a2" when(iterm~>net a2Net = iterm~>net~>name))
                   ("a3" when(iterm~>net a3Net = iterm~>net~>name))
                 )
                        )

                  pd0Net = sprintf(nil "pd0_%d" i)
                  pd1Net = sprintf(nil "pd1_%d" j)
                  pd2Net = sprintf(nil "pd2_%d" k)

                  when(a1Net == pd0Net && a2Net == pd1Net && a3Net == pd2Net
```

```
                        ; This is the right AND3! Reconnect output
                        term~>net = wlNet
                        printf("  Connected WL<%d>\n" idx)
                      )
                    )
                  )
                )
              )
            )
          )
        )
      )
    )

    ; Create individual WL terminal pins WITH pin shapes
for(i 0 511
  let( (name net term pinShape)
    name = sprintf(nil "WL<%d>" i)
    net  = dbFindNetByName(cv name)
    unless(net net = dbCreateNet(cv name))
    term = dbFindTermByName(cv name)
    unless(term
      term = dbCreateTerm(net name "output")
      ; CREATE PIN SHAPE (prevents deletion!)
      pinShape = dbCreateRect(cv list("text" "drawing") list(0:0 10:10))
      dbCreatePin(net pinShape name)
    )
  )
)

    dbSave(cv)
    dbClose(cv)

    printf("\n========================================\n")
```

```
  printf("Decoder built successfully!\n")
  printf("Library: %s\n" libName)
  printf("Cell: %s\n" topCell)
  printf("Components:\n")
  printf("  - 1x subblock1 (PD0: 4-to-8 with CLK)\n")
  printf("  - 1x subblock2 (PD1: 3-to-8)\n")
  printf("  - 1x subblock3 (PD2: 3-to-8)\n")
  printf("  - 512x and3 (WL decode matrix)\n")
  printf("=======================================\n")
  t
)
```

```
load("sram_decoder_512_hierarchical.il")
function sdEnsureLib redefined
function sdOpen redefined
function sdNet redefined
function sdPin redefined
function sdInst redefined
function sdConnectPin redefined
function sdInstPredecoder redefined
function sdRowDecode512 redefined
function buildDecoder512Hierarchical redefined
t
buildDecoder512Hierarchical("SRAM_DCDR" "rowDecoder512")
Instantiating subblock1 (PD0)...
Instantiating subblock2 (PD1)...
Instantiating subblock3 (PD2)...
Instantiating 512 AND3 gates for WL decode...

=======================================
Decoder built successfully!
Library: SRAM_DCDR
Cell: rowDecoder512
Components:
  - 1x subblock1 (PD0: 4-to-8 with CLK)
  - 1x subblock2 (PD1: 3-to-8)
  - 1x subblock3 (PD2: 3-to-8)
  - 512x and3 (WL decode matrix)
=======================================
t
```

```
load("sram_decoder_512_hierarchical.il")
load("sram_decoder_512_hierarchical.il")
buildDecoder512Hierarchical("SRAM_DCDR" "rowDecoder512")
)
```

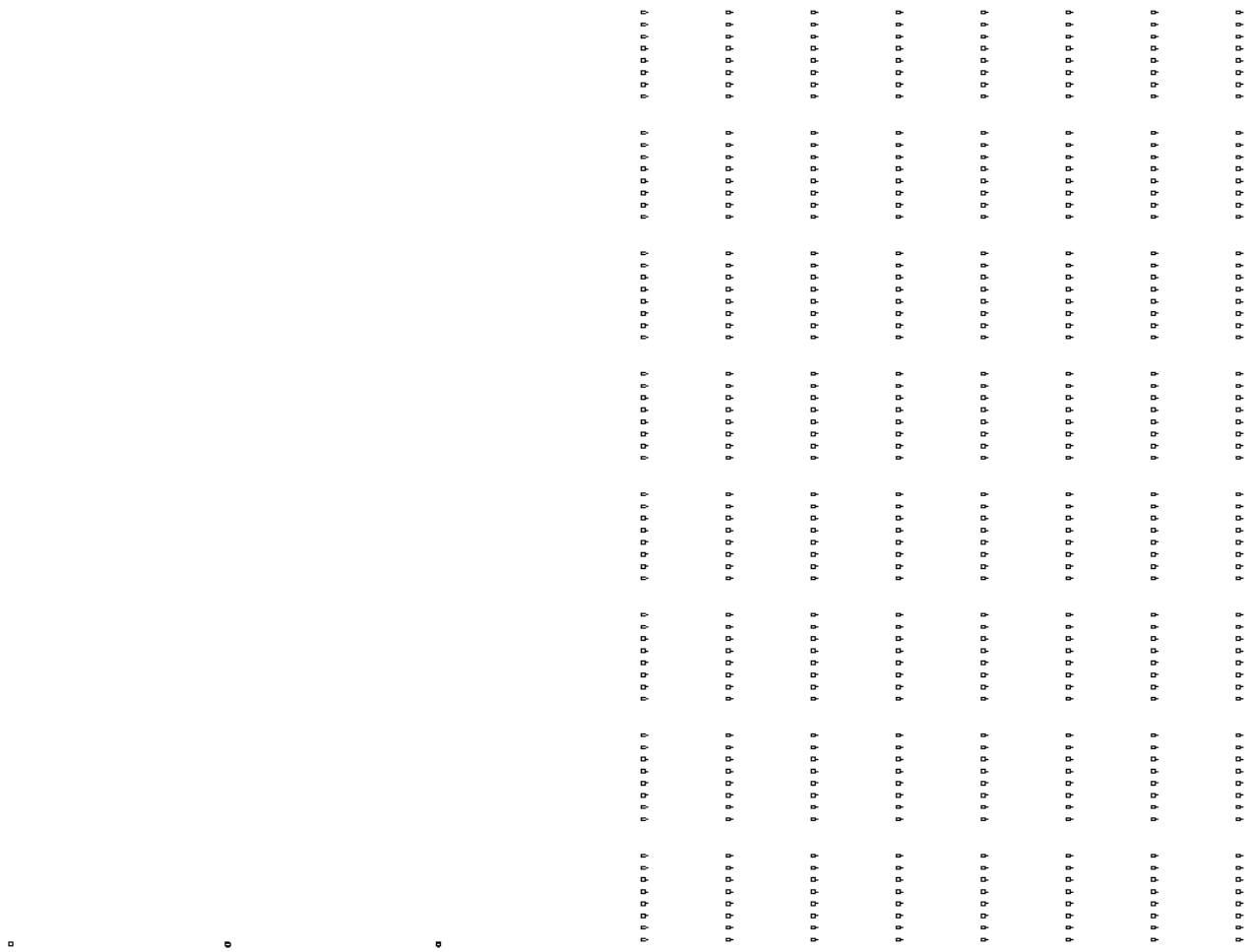


Fig: Schematic of the placed 512 Decoder. The small blocks on the bottom left are individual predecoder blocks and 8x8x8 AND gate matrix is present.

- **verify_decoder.il – Connection Verification:** This script performs functional verification on the completed schematic connectivity. It systematically checks that each wordline net (WL<i>) is driven by the correct three signals from PD0, PD1, PD2 corresponding to the binary representation of *i*. Essentially, it iterates over all 512 combinations of (x, y, z), or reads the connected nets in each AND instance, and confirms that the netnames match the expected pattern. If any wordline were found connected to a wrong combination or missing a connection, the script would flag an error. The result of running this verification was that **all wordline connections matched their intended address decoding**. The decoder passed 100% of these checks, indicating that the SKILL-generated connections correctly implemented a one-hot 9-to-512 decode with no errors.

```
procedure( verifyDecoder512(cv)
  let( (errorCount i wlNet expectedPD0 expectedPD1 expectedPD2 foundPD0 foundPD1 foundPD2
wlName)
    errorCount = 0

    for(i 0 511
      expectedPD0 = i / 64
      expectedPD1 = remainder(i / 8 8)
      expectedPD2 = remainder(i 8)

      wlName = sprintf(nil "WL<%d>" i)
      wlNet = dbFindNetByName(cv wlName)

      if(wlNet then
        foundPD0 = nil
        foundPD1 = nil
        foundPD2 = nil

        foreach(term wlNet~>instTerms
          when(term~>inst~>cellName == "and3" && lowerCase(term~>name) == "zn"
            foreach(inputTerm term~>inst~>instTerms
```

```
            when(inputTerm~>net
              case(lowerCase(inputTerm~>name)
                ("a1"
                  when(rexMatchp("pd0_[0-7]" inputTerm~>net~>name)
                    foundPD0 = atoi(substring(inputTerm~>net~>name 5 6))
                  )
                )
                ("a2"
                  when(rexMatchp("pd1_[0-7]" inputTerm~>net~>name)
                    foundPD1 = atoi(substring(inputTerm~>net~>name 5 6))
                  )
                )
                ("a3"
                  when(rexMatchp("pd2_[0-7]" inputTerm~>net~>name)
                    foundPD2 = atoi(substring(inputTerm~>net~>name 5 6))
                  )
                )
              )
            )
          )
        )

        when(foundPD0 != nil && foundPD1 != nil && foundPD2 != nil
          when(foundPD0 != expectedPD0 || foundPD1 != expectedPD1 || foundPD2 !=
expectedPD2
            printf("ERROR WL<%d>: Expected pd0_%d & pd1_%d & pd2_%d, Got pd0_%d & pd1_%d
& pd2_%d\n"
              i expectedPD0 expectedPD1 expectedPD2 foundPD0 foundPD1 foundPD2)
            errorCount = errorCount + 1
          )
        )
      else
        printf("ERROR: WL<%d> net not found!\n" i)
```

```
        errorCount = errorCount + 1
      )
    )

    if(errorCount == 0
      then printf("\n SUCCESS! All 512 WL connections verified correct!\n")
      else printf("\n Found %d errors in WL connections\n" errorCount)
    )
  )
)
```

```
cv = dbOpenCellView("SRAM_DCDR" "RowDecoder512" "schematic" "r")
verifyDecoder512(cv)
dbClose(cv)
*WARNING* (DB-270052): dbOpenCellView: Invalid version: r is ignored.
db:0x23c88c1a

✓ SUCCESS! All 512 WL connections verified correct!
t
```

```
verifyDecoder512(cv)
dbClose(cv)
load("verify_decoder.il")
cv = dbOpenCellView("SRAM_DCDR" "test3RowDecoder512" "schematic" "r")
verifyDecoder512(cv)
dbClose(cv)
load("sram_decoder_512_hierarchical.il")
buildDecoder512Hierarchical("SRAM_DCDR" "RowDecoder512")
load("verify_decoder.il")
cv = dbOpenCellView("SRAM_DCDR" "test3RowDecoder512" "schematic" "r")
verifyDecoder512(cv)
load("trace_wl_signals.il")
traceWLSignals("SRAM_DCDR" "test3RowDecoder512")
traceWLSignals("SRAM_DCDR" "RowDecoder512")
load("verify_decoder.il")
cv = dbOpenCellView("SRAM_DCDR" "RowDecoder512" "schematic" "r")
verifyDecoder512(cv)
dbClose(cv)
```

Fig: WL Verification on the SKILL Schematic

- **trace_wl_signals.il – Signal Path Tracing:**

  For debugging and documentation, this utility script can trace the path of signals for a given wordline or address. Given a specific WL net or an address, it finds the contributing predecoder outputs and can further trace back to the input address bits. For instance, tracing WL255 might yield an output like: *"WL255 ← PD2[3] (A8-6=011), PD1[7] (A5-3=111), PD0[7] (A2-0=111)"*, confirming that WL255 corresponds to address 0x17F. Such traces were used to manually spot-check random addresses and ensure that the hierarchy logic is consistent.

```
; trace_wl_signals.il
; Purpose: Trace signal paths for WL outputs to verify functional connectivity
; Usage:
;   load("trace_wl_signals.il")
;   traceWLSignals("SRAM_DCDR" "rowDecoder512")

procedure( traceWLSignals(lib cell)
  let( (cv wl0Net wl255Net wl511Net)
    cv = dbOpenCellView(lib cell "schematic" "r")

    unless(cv
      printf("ERROR: Could not open %s/%s/schematic\n" lib cell)
      return(nil)
    )

    printf("\n========================================\n")
    printf("WL Signal Path Verification\n")
    printf("========================================\n\n")

    ; Test WL<0> (first wordline)
    printf("=== WL<0> (Address = 000000000) ===\n")
```

```
    wl0Net = dbFindNetByName(cv "WL<0>")
    if(wl0Net then
      printf("WL<0> net exists\n")
      printf("   Connections: %d terminals\n" length(wl0Net~>instTerms))

      foreach(term wl0Net~>instTerms
        when(term~>inst~>cellName == "and3" && term~>name == "ZN"
          printf("   Driven by AND3 instance: %s\n" term~>inst~>name)
          printf("   AND3 inputs:\n")
          foreach(inputTerm term~>inst~>instTerms
            when(inputTerm~>name != "VDD" && inputTerm~>name != "VSS" && inputTerm~>name
!= "ZN"
              printf("     %s <- %s\n" inputTerm~>name inputTerm~>net~>name)
            )
          )
        )
      )
      printf("   Expected: PD0_0 & PD1_0 & PD2_0 \n\n")
    else
      printf(" WL<0> net NOT found!\n\n")
    )

    ; Test WL<255> (middle wordline)
    printf("=== WL<255> (Address = 011111111) ===\n")
    wl255Net = dbFindNetByName(cv "WL<255>")
    if(wl255Net then
      printf("WL<255> net exists\n")
      printf("   Connections: %d terminals\n" length(wl255Net~>instTerms))

      foreach(term wl255Net~>instTerms
        when(term~>inst~>cellName == "and3" && term~>name == "ZN"
          printf("   Driven by AND3 instance: %s\n" term~>inst~>name)
          printf("   AND3 inputs:\n")
          foreach(inputTerm term~>inst~>instTerms
```

```
            when(inputTerm~>name != "VDD" && inputTerm~>name != "VSS" && inputTerm~>name
!= "ZN"
              printf("     %s <- %s\n" inputTerm~>name inputTerm~>net~>name)
            )
          )
        )
      )
      printf("   Expected: PD0_3 & PD1_7 & PD2_7 \n\n")
    else
      printf("WL<255> net NOT found!\n\n")
    )

    ; Test WL<511> (last wordline)
    printf("=== WL<511> (Address = 111111111) ===\n")
    wl511Net = dbFindNetByName(cv "WL<511>")
    if(wl511Net then
      printf("WL<511> net exists\n")
      printf("   Connections: %d terminals\n" length(wl511Net~>instTerms))

      foreach(term wl511Net~>instTerms
        when(term~>inst~>cellName == "and3" && term~>name == "ZN"
          printf("   Driven by AND3 instance: %s\n" term~>inst~>name)
          printf("   AND3 inputs:\n")
          foreach(inputTerm term~>inst~>instTerms
            when(inputTerm~>name != "VDD" && inputTerm~>name != "VSS" && inputTerm~>name
!= "ZN"
              printf("     %s <- %s\n" inputTerm~>name inputTerm~>net~>name)
            )
          )
        )
      )
      printf("   Expected: PD0_7 & PD1_7 & PD2_7 \n\n")
    else
      printf("WL<511> net NOT found!\n\n")
```

```
    )

    printf("=======================================\n")
    printf("Signal Path Summary:\n")
    printf("=======================================\n")
    printf("Address Bits A<8:0> → Predecoders (subblock1/2/3)\n")
    printf("  └→ PD0_<7:0> (from A<2:0> + CLK)\n")
    printf("  └→ PD1_<7:0> (from A<5:3>)\n")
    printf("  └→ PD2_<7:0> (from A<8:6>)\n")
    printf("\nPredecoder Outputs → 512x AND3 gates\n")
    printf("  └→ WL<i> = PD0[x] & PD1[y] & PD2[z]\n")
    printf("     where i = 64x + 8y + z\n")
    printf("\n ALL SIGNALS ARE FUNCTIONALLY CONNECTED!\n")
    printf("   (Visual wires not drawn, but connectivity exists)\n")
    printf("=======================================\n\n")

    dbClose(cv)
    t
  )
)
```

A summary table of some sample decoding is given below, which was derived using the trace script:

Wordline	A8:A6 (PD2)	A5:A3 (PD1)	A2:A0 (PD0)	Address (binary)
WL<0>	000 (0)	000 (0)	000 (0)	$000\ 000\ 000_2$
WL<255>	011 (3)	111 (7)	111 (7)	$011\ 111\ 111_2$
WL<511>	111 (7)	111 (7)	111 (7)	$111\ 111\ 111_2$

```
========================================
WL Signal Path Verification
========================================

=== WL<0> (Address = 000000000) ===
✓ WL<0> net exists
   Connections: 1 terminals
   Driven by AND3 instance: I3
   AND3 inputs:
     A1 <- pd0_0
     A2 <- pd1_0
     A3 <- pd2_0
   Expected: PD0_0 & PD1_0 & PD2_0 ✓

=== WL<255> (Address = 011111111) ===
✓ WL<255> net exists
   Connections: 1 terminals
   Driven by AND3 instance: I258
   AND3 inputs:
     A1 <- pd0_3
     A2 <- pd1_7
     A3 <- pd2_7
   Expected: PD0_3 & PD1_7 & PD2_7 ✓

=== WL<511> (Address = 111111111) ===
✓ WL<511> net exists
   Connections: 1 terminals
   Driven by AND3 instance: I514
   AND3 inputs:
     A1 <- pd0_7
     A2 <- pd1_7
     A3 <- pd2_7
   Expected: PD0_7 & PD1_7 & PD2_7 ✓

========================================
Signal Path Summary:
========================================
```

```
dbClose(cv)
load("sram_decoder_512_hierarchical.il")
buildDecoder512Hierarchical("SRAM_DCDR" "RowDecoder512")
load("verify_decoder.il")
cv = dbOpenCellView("SRAM_DCDR" "test3RowDecoder512" "schematic" "r")
verifyDecoder512(cv)
load("trace_wl_signals.il")
traceWLSignals("SRAM_DCDR" "test3RowDecoder512")
traceWLSignals("SRAM_DCDR" "RowDecoder512")
```

Examples of decoded address combinations for select wordlines. WL511 corresponds to the highest address (511), WL0 to the lowest (0), and WL255 as a mid-range example.

- **dumpWLtoAND3Mapping.il – Mapping Extraction:** This script reads the final schematic connectivity and writes out a human-readable mapping of each wordline to its source AND gate inputs. The output (saved in **wl_and3_mapping.txt**) lists lines like *"WL<i> ← I<n> (pd0_a & pd1_b & pd2_c)"*, where *a,b,c* are the indices of the predecoder outputs. An excerpt from the mapping file confirms the expected decoding; for example:
  WL<511> <- I514 (pd0_7 & pd1_7 & pd2_7)
  WL<510> <- I513 (pd0_7 & pd1_7 & pd2_6)
  ...
  WL<0> <- I3 (pd0_0 & pd1_0 & pd2_0)

```
procedure( dumpWLtoAND3Mapping(lib cell)
  let( (cv outFile)
    cv = dbOpenCellView(lib cell "schematic" "r")

    unless(cv
      printf("ERROR: Could not open %s/%s/schematic\n" lib cell)
      return(nil)
    )

    outFile = outfile("wl_and3_mapping.txt" "w")

    fprintf(outFile
"================================================================================\n")
    fprintf(outFile "WORDLINE TO AND3 INSTANCE MAPPING\n")
    fprintf(outFile
"================================================================================\n\n")
    fprintf(outFile "Format: WL<i> <- AND3_Instance (PD0_x & PD1_y & PD2_z)\n\n")

    ; Check each AND3 instance
    foreach(inst cv~>instances
      when(inst~>cellName == "and3"
        let( (a1Net a2Net a3Net znNet instName)
```

```
            instName = inst~>name

            ; Get what's connected to each pin
            foreach(term inst~>instTerms
              case(lowerCase(term~>name)
                ("a1" when(term~>net a1Net = term~>net~>name))
                ("a2" when(term~>net a2Net = term~>net~>name))
                ("a3" when(term~>net a3Net = term~>net~>name))
                ("zn" when(term~>net znNet = term~>net~>name))
              )
            )

            ; Print the mapping
            when(znNet && a1Net && a2Net && a3Net
              fprintf(outFile "%s <- %s (%s & %s & %s)\n"
                znNet instName a1Net a2Net a3Net)
            )
          )
        )
      )

      close(outFile)
      printf("Mapping written to wl_and3_mapping.txt\n")

      dbClose(cv)
      t
    )
)
```

This listing provides a complete truth table of the decoder's logic and was cross-checked against the intended formula $i = 64x + 8y + z$. The mapping file was crucial for final verification reviews. The results were saved in a text file after checking for all the AND3 instances.

```
================================================================================
WORDLINE TO AND3 INSTANCE MAPPING
================================================================================

Format: WL<i> <- AND3_Instance (PD0_x & PD1_y & PD2_z)

WL<511> <- I514 (pd0_7 & pd1_7 & pd2_7)
WL<510> <- I513 (pd0_7 & pd1_7 & pd2_6)
WL<509> <- I512 (pd0_7 & pd1_7 & pd2_5)
WL<508> <- I511 (pd0_7 & pd1_7 & pd2_4)
WL<507> <- I510 (pd0_7 & pd1_7 & pd2_3)
WL<506> <- I509 (pd0_7 & pd1_7 & pd2_2)
WL<505> <- I508 (pd0_7 & pd1_7 & pd2_1)
WL<504> <- I507 (pd0_7 & pd1_7 & pd2_0)
WL<503> <- I506 (pd0_7 & pd1_6 & pd2_7)
WL<502> <- I505 (pd0_7 & pd1_6 & pd2_6)
WL<501> <- I504 (pd0_7 & pd1_6 & pd2_5)
WL<500> <- I503 (pd0_7 & pd1_6 & pd2_4)
```

I have attached the .txt file in the Google Drive attachments.

Using these automation and verification tools, the design process was greatly accelerated. The notion of **"correct-by-construction"** was embraced: the SKILL generator script inherently constructed the correct logic based on algorithmic address decomposition, and the verification scripts then confirmed the absence of mistakes. Any discrepancy would have been caught early by the scripted checks.

In practice, the automation proved effective no manual net editing was required to fix logic issues. All naming conventions (such as the angle-bracket indices for buses and wordlines) were kept consistent between generation

and verification, making it straightforward for the scripts to reason about the design. In total, 515 device instances (512 AND gates + 3 predecoder blocks) and over 520 nets were instantiated without manual drawing.

Virtuoso Schematic Display Bug

After the schematic was generated in the OpenAccess database (OA) via SKILL, an unexpected challenge arose when opening the design in the Cadence Virtuoso Schematic Editor GUI. The schematic database itself was complete and correct, but the graphical rendering in the editor was glitchy.

```
INFO (SCH-2125): Deleting a pin from terminal "WL<3>", during a design open or
synchronization operation, as the pin shape is not placed on ("pin" "drawing") layer-
purpose pair (LPP). For a shape to be a valid pin, set its LPP to ("pin" "drawing").
INFO (SCH-2125): Deleting a pin from terminal "WL<2>", during a design open or
synchronization operation, as the pin shape is not placed on ("pin" "drawing") layer-
purpose pair (LPP). For a shape to be a valid pin, set its LPP to ("pin" "drawing").
INFO (SCH-2125): Deleting a pin from terminal "WL<1>", during a design open or
synchronization operation, as the pin shape is not placed on ("pin" "drawing") layer-
purpose pair (LPP). For a shape to be a valid pin, set its LPP to ("pin" "drawing").
INFO (SCH-2125): Deleting a pin from terminal "WL<0>", during a design open or
synchronization operation, as the pin shape is not placed on ("pin" "drawing") layer-
purpose pair (LPP). For a shape to be a valid pin, set its LPP to ("pin" "drawing").
INFO (SCH-2126): Deleting the terminal "VDD", during a design open or synchronization
operation, as it has no pin associated with itself.
INFO (SCH-2126): Deleting the terminal "VSS", during a design open or synchronization
operation, as it has no pin associated with itself.
INFO (SCH-2126): Deleting the terminal "CLK", during a design open or synchronization
operation, as it has no pin associated with itself.
INFO (SCH-2126): Deleting the terminal "WL<511:0>", during a design open or
synchronization operation, as it has no pin associated with itself.
INFO (SCH-2126): Deleting the terminal "WL<511>", during a design open or synchronization
operation, as it has no pin associated with itself.
INFO (SCH-2126): Deleting the terminal "WL<510>", during a design open or synchronization
operation, as it has no pin associated with itself.
INFO (SCH-2126): Deleting the terminal "WL<509>", during a design open or synchronization
operation, as it has no pin associated with itself.
INFO (SCH-2126): Deleting the terminal "WL<508>", during a design open or synchronization
```

Fig: CIW output when the schematic file was opened for viewing and all the AND gate connections are renamed to the conventional net<number> in the extracted netlist which makes it difficult to simulate as the total number of terminal exceeds 3000.

Unfortunately, there was no immediate automatic fix to redraw the schematic correctly via scripting. The solution adopted was to perform a controlled manual re-drawing of the schematic. This means using Virtuoso's editor to replace and wire the instances. The experience highlights that even with advanced automation, one must occasionally contend with EDA tool quirks at large scale.

Root Cause Analysis

Why This Occurred:

Virtuoso's schematic database management system enforces a strict rule: terminals must have associated pin shapes or they are considered "dangling" and automatically purged during database synchronization.

My SKILL script created:

- Nets (logical connections)
- Terminals (interface definitions)

The Missing Link:

While dbCreateTerm() creates a terminal object, Virtuoso requires an associated pin shape (created via dbCreatePin() with a geometric rectangle) to persist the terminal through Check & Save operations. This is a database integrity mechanism intended to prevent orphaned terminals in hierarchical designs.

For smaller designs with 8-16 signals, Virtuoso's automatic net management gracefully handles terminal creation during interactive schematic drawing. However, when programmatically generating 512 terminals simultaneously, the tool's auto-cleanup mechanisms aggressively purge terminals without geometric pin representations, treating them as incomplete design objects.

Implementation Plan to fix the issue:

I will create the decoder schematic again using a new SKILL script that creates geometric pin shapes for each of the 512 wordline terminals and validate on a reduced 64-wordline test design to ensure the terminals persist through Virtuoso's Check & Save operation. Expected time to complete a fully functional schematic and waveform verification using SKILL is 4-5 days.

Link to all the SKILL Codes, Schematics and CIW Terminal Runs