

VL-401 Logic Synthesis

Running FSM minimization, state encoding, two-level and multi-level optimization

Dr. Bikram Paul • SCEE, IIT Mandi • Aug–Dec 2026

One program, `vl401.py`, runs the real Berkeley synthesis tools (SIS, STAMINA, NOVA, JEDI and espresso) on problems you write.

Topic	Command
FSM state minimization	<code>python3 vl401.py minimize FILE</code>
State encoding	<code>python3 vl401.py encode FILE</code>
Two-level (SOP) minimization	<code>python3 vl401.py twolevel FILE</code>
Multi-level optimization	<code>python3 vl401.py multilevel FILE</code>
Make bigger problems	<code>python3 vl401.py generate KIND</code>

What is in the folder

<code>vl401.py</code>	the program you run (needs only Python 3)
<code>bin/</code>	the synthesis tools (do not change)
<code>lab1_fsm_minimization/, lab2_...</code> <code>..., lab3_..., lab4_multi-</code> <code>level/</code>	one folder per topic and examples/
<code>my_problems/</code>	templates for every input format; put your own files here
<code>colab/</code>	a browser notebook, for Windows or macOS users who cannot use Linux or WSL
<code>source/</code>	how <code>bin/</code> was built (only needed on non-Intel/AMD Linux)

1 Get it running on Linux

The kit is made for **Linux**. Everything in this guide is typed in a Linux terminal. *Only have a Windows PC? Do Section 2 first, then come back here.*

1.1 What you need

- 64-bit Linux on an Intel/AMD processor. Check with `uname -m`: it must print `x86_64`. Any distribution works (Ubuntu, Debian, Mint, Fedora, Arch, ...), including the institute lab machines.
- `python3` and `unzip`. Most distributions already have both. If not:

Distribution	Command
Ubuntu, Debian, Mint, WSL	<code>sudo apt install python3 unzip</code>
Fedora, RHEL, Rocky	<code>sudo dnf install python3 unzip</code>
Arch, Manjaro	<code>sudo pacman -S python unzip</code>

No other packages, no compiler and no administrator rights are needed for the kit itself.

1.2 Install and test

1. Open a terminal (Ubuntu: **Ctrl+Alt+T**).
2. Unzip the kit into your home folder and enter it:

```
cd ~
unzip ~/Downloads/VL401_Synthesis_Lab.zip
cd VL401_Synthesis_Lab
```

3. Test the tools:

```
python3 vl401.py check
```

What a working installation prints

```
=====
CHECKING THE TOOLS
=====
```

```
Python   : 3.13.16 on Linux x86_64
sis      : ~/VL401_Synthesis_Lab/bin/sis
espresso : ~/VL401_Synthesis_Lab/bin/espresso
stamina  : ~/VL401_Synthesis_Lab/bin/stamina
nova     : ~/VL401_Synthesis_Lab/bin/nova
jedi     : ~/VL401_Synthesis_Lab/bin/jedi
espresso test: a+b -> 2 products  OK
stamina test : 3 states -> 2  OK
nova test   : OK
sis_lib     : OK
```

```
All tools work. Try:  python3 vl401.py minimize lab1_fsm_minimization/examples/detector_010...
```

4. Run your first lab:

```
python3 vl401.py minimize lab1_fsm_minimization/examples/detector_010_110.fsm
```

5. *Optional shortcut*: to type `vl401` instead of `python3 vl401.py` from any folder, run this once, then open a new terminal:

```
echo "alias vl401='python3 $HOME/VL401_Synthesis_Lab/vl401.py'" >> ~/.bashrc
```

Afterwards, for example: `vl401 twolevel my_circuit.sop`.

1.3 The terminal commands you need

Command	What it does
<code>pwd</code>	show which folder you are in
<code>ls, ls examples</code>	list the files here, or in <code>examples</code>
<code>cd lab3_two_level / cd .. / cd ~/VL401_Synthesis_Lab</code>	go into a folder / one level up / back to the kit
Tab key	completes file names: type <code>lab3</code> and press Tab
Up arrow	repeat or edit the previous command
<code>cp my_problems/template.fsm my_problems/mine.fsm</code>	copy a template to start your own problem
<code>gedit FILE &, nano FILE, code FILE</code>	edit a file (graphical editor, terminal editor, VS Code)
<code>python3 vl401.py ... > result.txt</code>	save the output to a file instead of the screen
<code>python3 vl401.py ... less</code>	read a long output page by page (q to quit)
Ctrl+C	stop a command that takes too long

If something goes wrong on Linux

Message	Fix
<code>python3: command not found</code>	install it (table above)
<code>unzip: command not found</code>	install it (table above), or right-click the zip in the file manager and choose <i>Extract Here</i>
<code>file not found: ...</code>	you are in the wrong folder: <code>cd ~/VL401_Synthesis_Lab</code> , or give the full path of the file
<code>Permission denied on bin/</code>	<code>chmod +x bin/*</code> (the script normally does this itself)
<code>... are for Intel/AMD (x86-64) Linux</code>	ARM machine (e.g. Raspberry Pi): rebuild once with <code>bash source/build_from_source.sh</code>
<code>an ERROR: about your file</code>	the message names the line; compare with the templates in <code>my_problems/</code>

2 For Windows PC

The tools are Linux programs, so on Windows you run them inside Linux. There are two ways.

Way	When to use it
1. WSL (recommended)	your own Windows 10 (version 2004 or newer) or Windows 11 PC. You get a real Ubuntu inside Windows and then follow Section 1 exactly.
2. Google Colab	you cannot install WSL (managed or very old PC), or you use macOS, a Chromebook or a tablet. Runs in the web browser; nothing is installed.

2.1 Way 1: WSL (Windows Subsystem for Linux)

1. Click *Start*, type **PowerShell**, right-click it and choose *Run as administrator*. Type:

```
wsl --install
```

Restart the PC when it asks.

2. After the restart, an *Ubuntu* window opens (or start *Ubuntu* from the Start menu). Choose a Linux user name and password. While you type the password nothing is shown; that is normal.
3. In the Ubuntu window, install `unzip` and copy the kit from your Windows *Downloads* folder (replace `YOUR_NAME` with your Windows user name;
`ls /mnt/c/Users` lists them):

```
sudo apt update && sudo apt install -y unzip
cp /mnt/c/Users/YOUR_NAME/Downloads/VL401_Synthesis_Lab.zip ~
```

4. Now continue with Section 1.2, “Install and test”. Every command in this guide works unchanged.

Working comfortably in WSL

- Keep the kit in your Linux home folder (`~`), not under `/mnt/c`: it is faster there and the file permissions are right.
- `explorer.exe .` opens the current Linux folder in Windows Explorer, so you can see and copy files.
- Edit files with Windows Notepad (`notepad.exe my_problems/mine.fsm`) or VS Code with the *WSL* extension (`code .`). Files saved by Windows editors are fine: the program accepts Windows line endings.
- If `wsl --install` reports that virtualization is disabled, enable *Intel VT-x* or *AMD-V / SVM* in the BIOS setup, or use Way 2.

2.2 Way 2: Google Colab (browser only)

1. Go to `colab.research.google.com` and sign in with any Google account.
2. *File* → *Upload notebook* and choose `colab/VL401_Synthesis_Lab.ipynb` from the kit (unzip it on Windows first: right-click the zip, *Extract All*).
3. Run the first code cell (click it, press **Shift+Enter**). When asked, choose the file `VL401_Synthesis_Lab.zip`. It ends with the same “All tools work” message.
4. Each lab has a cell starting with `%%writefile`: edit the problem there, run it, then run the cell below it. Colab runs Linux, so the commands are the same as in this guide, with a `!` in front, e.g. `!python3 vl401.py minimize my.fsm`.
5. Files disappear when the Colab session ends; the last cell of the notebook downloads your `my_problems` folder.

3 FSM state minimization

Question it answers: which states of my machine can never be told apart by any input sequence, and what is the smallest equivalent machine?

Input file: state table .fsm (example: lab1_fsm_minimization/examples/detector_010_110.fsm)

```
# Slides 26-28: output 1 when a 3-bit block was 010 or 110      <- '#' starts a comment
inputs: 0 1              <- one column per input value (two input bits: 00 01 10 11)
reset: S0                <- optional; default = first state
S0  S1/0  S2/0          <- state, then NEXT/OUTPUT for input 0, then for input 1
S1  S3/0  S4/0
S2  S5/0  S6/0
S3  S0/0  S0/0
S4  S0/1  S0/0          <- in S4, input 0 gives output 1
S5  S0/0  S0/0
S6  S0/1  S0/0          <- write '-' for a cell that is not specified
```

```
python3 vl401.py minimize lab1_fsm_minimization/examples/detector_010_110.fsm
```

Output (complete)

```
>> Goal: find states that no input sequence can tell apart (equivalent
>> states) and merge them. Fewer states = fewer flip-flops and smaller next-
>> state logic.
```

```
-----
STEP 1  The machine you gave
-----
```

state	in=0	in=1
S0	S1/0	S2/0
S1	S3/0	S4/0
S2	S5/0	S6/0
S3	S0/0	S0/0
S4	S0/1	S0/0
S5	S0/0	S0/0
S6	S0/1	S0/0

```
states = 7, input bits = 1, output bits = 1, reset state = S0
-----
```

```
STEP 2  Partitioning method (the lecture method), done by this script
-----
```

```
Pass 1: put states with the same OUTPUTS for every input in one group.
```

state	outputs for inputs 0 1
S0	0 0
S1	0 0
S2	0 0
S3	0 0
S4	1 0
S5	0 0
S6	1 0

```
P1 = {S0,S1,S2,S3,S5} {S4,S6}    (2 groups)
```

```
Pass 2: inside each group, compare which GROUP each next state is in (the signature).
```

```
Group names in P1: A={S0,S1,S2,S3,S5} B={S4,S6}
```

state	group	input->next(group)	signature
S0	A	0->S1(A) 1->S2(A)	(A,A)
S1	A	0->S3(A) 1->S4(B)	(A,B)
S2	A	0->S5(A) 1->S6(B)	(A,B)
S3	A	0->S0(A) 1->S0(A)	(A,A)
S5	A	0->S0(A) 1->S0(A)	(A,A)
S4	B	0->S0(A) 1->S0(A)	(A,A)
S6	B	0->S0(A) 1->S0(A)	(A,A)

```
P2 = {S0,S3,S5} {S1,S2} {S4,S6} (3 groups)
```

Pass 3: inside each group, compare which GROUP each next state is in (the signature).

Group names in P2: A={S0,S3,S5} B={S1,S2} C={S4,S6}

state	group	input->next(group)	signature
S0	A	0->S1(B) 1->S2(B)	(B,B)
S3	A	0->S0(A) 1->S0(A)	(A,A)
S5	A	0->S0(A) 1->S0(A)	(A,A)
S1	B	0->S3(A) 1->S4(C)	(A,C)
S2	B	0->S5(A) 1->S6(C)	(A,C)
S4	C	0->S0(A) 1->S0(A)	(A,A)
S6	C	0->S0(A) 1->S0(A)	(A,A)

```
P3 = {S0} {S3,S5} {S1,S2} {S4,S6} (4 groups)
```

Pass 4: inside each group, compare which GROUP each next state is in (the signature).

Group names in P3: A={S0} B={S3,S5} C={S1,S2} D={S4,S6}

state	group	input->next(group)	signature
S0	A	0->S1(C) 1->S2(C)	(C,C)
S3	B	0->S0(A) 1->S0(A)	(A,A)
S5	B	0->S0(A) 1->S0(A)	(A,A)
S1	C	0->S3(B) 1->S4(D)	(B,D)
S2	C	0->S5(B) 1->S6(D)	(B,D)
S4	D	0->S0(A) 1->S0(A)	(A,A)
S6	D	0->S0(A) 1->S0(A)	(A,A)

```
P4 = {S0} {S3,S5} {S1,S2} {S4,S6} (4 groups)
```

No group split, so P4 = P3 and the method STOPS.

```
>> Equivalent states (each line becomes ONE state):
```

```
S3 = S5
```

```
S1 = S2
```

```
S4 = S6
```

```
-----
STEP 3 STAMINA (Berkeley SIS) minimizes the same machine
-----
```

```
Number of states in original machine : 7
```

```
Number of states in minimized machine : 4
```

```
Minimized machine, built from the groups of step 2
```

```
(a merged state is named after its members, e.g. S1_S2 = S1 and S2 merged):
```

state	in=0	in=1
S0	S1_S2/0	S1_S2/0
S1_S2	S3_S5/0	S4_S6/0
S3_S5	S0/0	S0/0
S4_S6	S0/1	S0/0

```
-----
STEP 4 Summary
-----
```

machine	states	flip-flops (binary code)
---------	--------	--------------------------

original	7	3
----------	---	---

minimized	4	2
-----------	---	---

```
>> The lecture method (step 2) and STAMINA (step 3) give the same number of
>> states.
```

How to read it

1. **STEP 1** echoes your table. Check it first: a typo in the file shows up here.
2. **STEP 2** is exactly the board method. *Pass 1* groups states with the same outputs. In every later pass, each state gets a *signature*: the group of its next state for each input. States in one group with different signatures are split. When a pass splits nothing, the method stops.
3. **Equivalent states** lists which states become one. Here: S3=S5, S1=S2, S4=S6.
4. **STEP 3** runs STAMINA, the Berkeley tool, as an independent check, and prints the reduced machine. A merged state is named after its members (S1_S2); in big machines they are called M1, M2, ... with a list.
5. **STEP 4**: 7 states needed 3 flip-flops, 4 states need 2.

Bigger problems

`big_history_1101011.fsm` has 64 states; `big_random_40state.fsm` has 40 states with 2 input and 2 output bits. For big machines use `--brief`, which prints only the group sizes per pass. Make your own:

```
python3 vl401.py generate history 10110111 -o my_problems/h.fsm
python3 vl401.py generate copies --states 10 --copies 6 \
    --ibits 2 --obits 2 -o my_problems/c.fsm
python3 vl401.py minimize my_problems/h.fsm --brief --save my_problems/h_min.ki
```

`history P` builds a deliberately wasteful machine that remembers the last (`length-1`) inputs, so most states are equivalent. `copies` copies every state of a random machine, so the minimum is known in advance. `--save` writes the reduced machine for Lab 2.

Part of the output for the 64-state machine (`--brief`)

```
Pass 1: put states with the same OUTPUTS for every input in one group.
P1: 2 groups, sizes 63 1
Pass 2: inside each group, compare which GROUP each next state is in (the signature).
P2: 3 groups, sizes 61 2 1
Pass 3: inside each group, compare which GROUP each next state is in (the signature).
P3: 4 groups, sizes 57 4 2 1
Pass 4: inside each group, compare which GROUP each next state is in (the signature).
P4: 5 groups, sizes 49 8 4 2 1
Pass 5: inside each group, compare which GROUP each next state is in (the signature).
P5: 6 groups, sizes 33 16 8 4 2 1
Pass 6: inside each group, compare which GROUP each next state is in (the signature).
P6: 7 groups, sizes 22 11 16 8 4 2 1
Pass 7: inside each group, compare which GROUP each next state is in (the signature).
P7: 7 groups, sizes 22 11 16 8 4 2 1
No group split, so P7 = P6 and the method STOPS.
>> Equivalent states (each line becomes ONE state):
    22 states: H000000 = H000010 = H000100 = H001000 = H001010 ... H111100
    11 states: H000001 = H000101 = H001001 = H010001 = H010101 ... H111001
    16 states: H000011 = H000111 = H001011 = H001111 = H010011 ... H111111
    H000110 = H001110 = H010110 = H011110 = H100110 = H101110 = H110110 = H111110
    H001101 = H011101 = H101101 = H111101
    H011010 = H111010
STEP 4 Summary
-----
machine      states  flip-flops (binary code)
-----
original     64      6
minimized    7       3
>> The lecture method (step 2) and STAMINA (step 3) give the same number of
>> states.
```

Try this

1. Run `moore_toggle.fsm` (slide 39). Row matching cannot merge S0 and S2. Which pass of the partitioning method shows that they are equivalent?
2. Run `exampleC_4state.fsm` (slide 38). S1 and S2 do not have the same next states. Why are they still merged?
3. Generate history machines for patterns of length 4, 5, 6 and 7. How does the number of passes relate to the pattern length?
4. Run `incomplete_rules_example.fsm`. Why is STEP 2 skipped?

4 State encoding

Question it answers: which binary code should each state get so that the next-state and output logic is small?

Input: the same .fsm or .kiss2 file as Lab 1. Example (slides 67–69):

lab2_state_encoding/examples/example3_ABCD.fsm

```
inputs: 0 1
reset: A
A  C/0  D/0
B  C/0  A/0
C  B/0  D/0
D  A/1  B/1
```

```
python3 vl401.py encode lab2_state_encoding/examples/example3_ABCD.fsm \
--codes A=00,B=01,C=10,D=11 --exhaustive --show your
```

Output (from STEP 2)

4 states need 2 code bits (4 codes, 0 unused; unused codes become don't-cares).

STEP 2 Which pairs SHOULD get adjacent codes (lecture rules, slides 63-66)

```
>> Adjacent codes differ in one bit. Rule 1: same next state on the same
>> input. Rule 2: next states of one state under adjacent inputs. Rule 3:
>> same output on the same input. MUSTANG weight W = k*(common next states) +
>> (shared output 1s), slides 73-75.
```

pair	rule 1	rule 2	rule 3	MUSTANG W
A-B	1	1	2	2
A-C	1	1	2	2
B-D		1		2
C-D		1		2
B-C			2	0
A-D				0

STEP 3 Cost of each encoding (espresso, two-level, multi-output)

encoding	products	literals	per-output lits	codes
binary	5	10	12	A=00 B=01 C=10 D=11
gray	6	14	16	A=00 B=01 C=11 D=10
onehot	6	14	25	A=1000 B=0100 C=0010 D=0001
NOVA (SIS)	5	11	12	A=10 B=11 C=00 D=01
JEDI (SIS)	5	11	12	A=11 B=10 C=01 D=00
your codes	5	10	12	A=00 B=01 C=10 D=11

```
>> products / literals: one PLA for all outputs, a product shared by several
>> outputs counted once.
>> per-output lits: every output minimized on its own and the literals added
>> up (the count used on slides 50-53). The two counts can prefer DIFFERENT
>> encodings.
>> Best here by shared products: binary (5 products). Best by per-output
>> literals: binary (12).
```

STEP 4 Try EVERY encoding with 2 bits (the brute-force answer)

products	literals	per-output lits	how many encodings
5	10	12	4
5	11	12	4
5	12	13	2
5	13	16	2
6	13	13	6
6	14	16	6

```

best by shared products   : A=00 B=01 C=10 D=11   (5 products, 10 literals)
best by per-output literals: A=00 B=01 C=10 D=11   (12 literals)
>> Compare the heuristics in step 3 with these best possible values.

```

```

-----
STEP *   Logic for encoding 'your codes'
-----

```

```

inputs: x (input), y0 y1 (present-state bits); outputs: Y0+ Y1+ z
Y0+    = x' y0' + x y1'
Y1+    = x y0 + x y1' + y0 y1'
z      = y0 y1
>> A product that appears in several equations is built once.

```

How to read it

1. **STEP 2** is the lecture analysis. For each pair of states: how often Rule 1, 2 and 3 ask for adjacent codes (slides 63–66), and the MUSTANG weight W (slides 73–75). Pairs at the top should get adjacent codes.
2. **STEP 3** compares encodings: binary, Gray, one-hot, the SIS heuristics NOVA and JEDI, and yours (`--codes`). For each it builds the encoded logic and minimizes it with espresso.
 - **products** = AND gates (PLA rows). A product used by several outputs is counted once.
 - **literals** = inputs to those AND gates.
 - **per-output lits** = each output minimized on its own and the literals added up. This is how slides 50–53 count.
3. **STEP 4** (`--exhaustive`) tries every encoding. It shows how many encodings give each cost, so you can judge how close the heuristics come to the best. Here the slide-68 encoding A=00, B=01, C=10, D=11 is the best (5 products), and the Gray order (the “random” encoding of slide 69) gives 6.
4. **STEP *** (`--show your`) prints the logic: $Y0^+$, $Y1^+$ are the next-state bits, z the output, $y0 y1$ the present-state bits, x the input.

Different cost counts can choose different encodings

Run `exampleC_3state.fsm` with `--exhaustive` (the 3-state machine of slides 49–53 and 73–77):

- counting **per-output literals** (as on slide 53): S1 adjacent to both others is best, with 11 literals;
- counting **shared products** (one PLA): S4 in the middle is best, with 3 products, and that is NOVA's choice.

Neither is “wrong”: the best encoding depends on how the logic will be built.

exampleC_3state.fsm --exhaustive (from STEP 3)

```

STEP 3   Cost of each encoding (espresso, two-level, multi-output)
-----

```

encoding	products	literals	per-output lits	codes
binary	5	11	11	S1=00 S3=01 S4=10
gray	5	10	12	S1=00 S3=01 S4=11
onehot	6	11	18	S1=100 S3=010 S4=001
NOVA (SIS)	3	5	12	S1=11 S3=00 S4=10
JEDI (SIS)	5	9	11	S1=10 S3=11 S4=00

```

>> products / literals: one PLA for all outputs, a product shared by several
>> outputs counted once.
>> per-output lits: every output minimized on its own and the literals added
>> up (the count used on slides 50–53). The two counts can prefer DIFFERENT
>> encodings.
>> Best here by shared products: NOVA (SIS) (3 products). Best by per-output
>> literals: binary (11).

```

```

-----
STEP 4   Try EVERY encoding with 2 bits (the brute-force answer)
-----

```

```

trying 24 encodings (about 1 s) ...

```

products	literals	per-output lits	how many encodings
3	5	12	2
4	9	11	2
4	9	12	2
5	8	11	2
5	9	11	2
5	9	12	4
5	10	12	6
5	11	11	2
6	12	12	2

```

best by shared products   : S1=11 S3=00 S4=01   (3 products, 5 literals)
best by per-output literals: S1=00 S3=01 S4=10   (11 literals)
>> Compare the heuristics in step 3 with these best possible values.

```

Bigger problems

Exhaustive search is only possible for small machines: 7 states in 3 bits already have $8!/1! = 40,320$ encodings, and the program refuses above 10,000 (change with `--max-tries`). That is exactly why heuristics exist. Minimize a big machine in Lab 1, save it, then compare the heuristics on it:

```

python3 vl401.py minimize lab1_fsm_minimization/examples/big_random_40state.fsm
                    --save my_problems/r8.kiss2
python3 vl401.py encode my_problems/r8.kiss2 --multilevel

```

`--multilevel` adds a column with the literal count after multi-level optimization (Lab 4).

Try this

1. In Example 3, which pairs does STEP 2 put at the top? Are they adjacent in $A=00, B=01, C=10, D=11$?
2. Try `--codes A=00, B=01, C=11, D=10`. Which demanded pairs end up on the diagonal of the 2-bit square?
3. Why does one-hot need so many literals here, but is popular in FPGAs anyway?

5 Two-level (SOP) minimization

Question it answers: what is the smallest AND–OR (sum of products) for each output? This is what you do with K-maps, done by espresso, the standard tool, for any number of inputs.

Input file: equations .sop (slide notation)

```
inputs: A B C D          <- needed for m(...); first = most significant bit
F = ab + a'c + bcd'      <- side by side = AND, + = OR, ' = NOT (or !a)
G = z(x(uv + w) + u'y')  <- brackets are allowed
a = m(0,2,3,5,6,7,8,9) + d(10,11,12,13,14,15) <- minterms + don't cares
```

Variable names are one letter, optionally followed by digits: a, B, x1, a12. For longer names use the .eqn format (Lab 4). Big truth tables can be given as espresso .pla files (template in my_problems/).

```
python3 vl401.py twolevel lab3_two_level/examples/bcd_segment_a.sop
```

Output (from STEP 1)

STEP 1 The function as a truth table / PLA (what espresso reads)

```
inputs : A B C D
outputs: a
14 input rows given (1 = ON, - = don't care, rows not listed = 0)
0000 1
0010 1
0011 1
0101 1
0110 1
0111 1
1000 1
1001 1
1010 -
1011 -
1100 -
1101 -
1110 -
1111 -
```

STEP 2 Minimized by espresso

Result as a PLA (each row = one AND gate; output columns say which ORs use it):

```
-0-0 1    = B'D'
1--- 1    = A
--1- 1    = C
-1-1 1    = BD
```

a = B'D' + A + C + BD

K-map of a (rows AB, columns CD, Gray order)

AB\CD	00	01	11	10
00	1:1	.	1:3	1:1,3
01	.	1:4	1:3,4	1:3
11	X:2	X:2,4	X:2,3,4	X:2,3
10	1:1,2	1:2	X:2,3	X:1,2,3

product #1 = B'D'

product #2 = A

product #3 = C

product #4 = BD

(1:1,3 = a 1-cell covered by products #1 and #3; X = don't care; . = 0)

STEP 3 Summary

products	literals
1	4
2	1
3	1
4	2

```

truth table (one product per input row)  14      56
minimized by espresso                   4       6
>> A product used by several outputs is counted once (it is one AND gate).
>> Literals = total inputs to the AND gates.

```

How to read it

1. **STEP 1:** the truth table espresso reads. 1 = ON, - = don't care; rows that are not listed are 0.
2. **STEP 2:** each result row such as $-0-0 \ 1 = B'D'$ is one AND gate. In $-0-0$, the positions are the inputs in order (here $A B C D$): 0 = complemented, 1 = true, - = not used. The output bits say which outputs use that gate.
3. **K-map** (4 inputs or fewer): 1:1, 3 is a 1-cell covered by products #1 and #3, X is a don't care, . is 0. Draw a loop for each product and you have the K-map solution.
4. **STEP 3:** size before and after. Here 14 truth-table rows become 4 products with 6 literals: $a = B'D' + A + C + BD$.

Bigger problems: watch two-level logic explode

```

python3 vl401.py generate parity 10 -o my_problems/p10.sop      # 512 products
python3 vl401.py generate compare 5 -o my_problems/cmp5.sop    # a > b
python3 vl401.py generate adder 4 -o my_problems/add4.sop
python3 vl401.py twolevel my_problems/add4.sop

```

Add `--exact` for a guaranteed minimum (slower). Example: a 3-bit adder needs 31 products and 116 literals as a two-level circuit. Compare with Lab 4.

Try this

1. Run `bcd_segment_a.sop` again without the `+ d(...)` part. How many products now?
2. Run `parity4.sop` (slide 80). Why does every 1 of the K-map stay alone?
3. Run `example3_encoded.pla` (slide 68). Which product is shared by two outputs?

6 Multi-level optimization

Question it answers: how much smaller does the circuit get when we factor expressions and share common sub-expressions (kernels and cubes) between outputs?

Input: `.sop` (same notation as Lab 3, but no `m(...)`) or `.eqn`. Example:

`lab4_multi_level/examples/slide92_FG.sop`

$$F = ae + be + cde + ab$$

$$G = ad + ae + be + bc$$

```
python3 vl401.py multilevel lab4_multi_level/examples/slide92_FG.sop
```

Output (from STEP 1)

STEP 1 The input (as SIS reads it) and its size

$$F = ab + ae + be + cde$$

$$G = ad + ae + be + bc$$

literals as sum of products = 17, as best factored form = 14
 >> lits(sop) counts every literal in the SOP. lits(fac) counts literals after
 >> factoring each output on its own (no sharing between outputs).

STEP 2 Factored form of each output (slide 79)

$$F = e(cd + b + a) + ab$$

$$G = e(b + a) + ad + bc$$

STEP 3 Kernels and co-kernels of each output (slides 85-93)

>> co-kernel = the cube taken outside the bracket; kernel = what stays inside
 >> (cube-free, at least 2 terms). Level 0 = nothing more inside can be
 >> factored.

F = ab + ae + be + cde		
co-kernel(s)	kernel	level
a	b + e	0
b	a + e	0
e	a + b + cd	0
1	ab + ae + be + cde	1

G = ad + ae + bc + be		
co-kernel(s)	kernel	level
a	d + e	0
e	a + b	0
b	c + e	0
1	ad + ae + bc + be	1

STEP 4 Kernel extraction: share the best common kernel (gkx, slides 89-104)

$$F = t2e + ab + cde$$

$$G = t2e + ad + bc$$

$$t2 = a + b$$

>> Any new node `t..` is a multi-cube divisor found in the intersection of
 >> kernels (Brayton-McMullen). If nothing changed, no shared kernel saves
 >> literals.

STEP 5 Cube extraction: share the best common cube (gcx, slides 105-108)

$$F = ab + ae + be + cde$$

$$G = ad + ae + be + bc$$

>> A cube of `k` literals used `r` times saves `r*k - r - k` literals. SIS only
 >> extracts when the saving is positive.

STEP 6 Full SIS optimization script (script.rugged)

```

F = G e + a b + c d e
G = a d + a e + b c + b e
>> script.rugged combines extraction, resubstitution and simplification; it
>> is what you would run on a real design.

```

STEP 7 Summary

network	literals (sop)
input (SOP)	17
after kernel extraction (gkx)	15
after cube extraction (gcx)	17
after script.rugged	15

```

>> Each step above starts again from the input, so you can compare them.
>> Fewer literals = smaller circuit (in static CMOS, transistors are roughly
>> 2 x literals).

```

How to read it

Step	Meaning
1. input	<code>lits(sop)</code> : literals as a sum of products (17). <code>lits(fac)</code> : after factoring each output on its own (14).
factored form	the best bracketed form of each output, as on slide 79.
kernels	co-kernel = the cube taken <i>outside</i> the bracket; kernel = what stays <i>inside</i> . Level 0 = nothing inside can be factored further (slides 85–93).
gkx (kernel extraction)	a new node $t_2 = a + b$, used by both F and G. This is the intersection of the kernels $a + b + cd$ and $a + b$ (Brayton–McMullen, slides 89–92). 17 → 15 literals.
gcx (cube extraction)	shares a common <i>product</i> . Nothing happens here: no cube saves literals ($rk - r - k \leq 0$, slide 108).
script.rugged	SIS's standard full optimization. It even writes $F = Ge + ab + cde$: a Boolean substitution ($Ge = ae + be$) that pure algebraic division cannot find.
summary	literals after each step; every step starts again from the input.

New shared nodes are called $t_1, t_2, \dots$

Input file: SIS equations .eqn (for real signal names; example eqn_format_example.eqn)

```

INORDER = req ack busy en1 en2;          <- inputs
OUTORDER = grant1 grant2;                <- outputs
grant1 = req*!busy*en1 + req*ack*en1;    <- * = AND, + = OR, ! = NOT, ';' at the end
grant2 = req*!busy*en2 + req*ack*en2;

```

Bigger problems

```

python3 vl401.py multilevel lab4_multi_level/examples/big_adder4.sop \
--steps factor,rugged
python3 vl401.py generate random --inputs 12 --outputs 6 --cubes 12 \
--seed 4 -o my_problems/r.sop
python3 vl401.py multilevel my_problems/r.sop --steps gkx,gcx,rugged

```

`--steps` chooses the steps (factor, kernels, gkx, gcx, rugged). The 4-bit adder: 398 literals as a sum of products, 54 after `script.rugged`.

Summary for the 4-bit adder

 STEP 3 Summary

network	literals (sop)
-----	-----
input (SOP)	398
after script.rugged	54

>> Each step above starts again from the input, so you can compare them.
 >> Fewer literals = smaller circuit (in static CMOS, transistors are roughly
 >> 2 x literals).

Try this

1. Run `slide88_kernels.sop` with `--steps kernels` and compare with the table on slide 88.
2. Run `slide105_cube_extraction.sop` and `cube_extraction_pays.sop`. Why does `gcx` do nothing in the first and extract yz in the second? Use the formula $r\bar{k} - r - k$.
3. Run `parity4.sop`. Which structure does `script.rugged` find, and how many literals does it save?

7 Writing your own problems: format summary

Format	Labs	Rules
.fsm	1, 2	inputs: 0 1, optional reset: A, then STATE NEXT/OUT NEXT/OUT ...; - = not specified
.kiss2	1, 2	header .i .o .s .p .r; rows INPUT PRESENT NEXT OUTPUT; .e at the end
.sop	3, 4	name = expression; side by side = AND, + = OR, ' or ! = NOT, brackets; m(...) + d(...) needs an inputs: line (Lab 3 only)
.pla	3	.i .o .ilb .ob, rows INPUTCUBE OUTPUTS; - = either value / don't care
.eqn	3, 4	INORDER = ...; OUTORDER = ...; then out = a*!b + c;

Templates with comments: `my_problems/template.*`. Lines starting with # are comments in every format.

To make ...	Command
a big redundant FSM	<code>python3 vl401.py generate history 10110111 -o f.fsm</code>
a random FSM, known minimum	<code>python3 vl401.py generate copies --states 8 --copies 5 -o f.fsm</code>
an n -bit adder	<code>python3 vl401.py generate adder 4 -o f.sop</code>
n -input parity	<code>python3 vl401.py generate parity 8 -o f.sop</code>
an n -bit comparator	<code>python3 vl401.py generate compare 4 -o f.sop</code>
a random multi-output function	<code>python3 vl401.py generate random --inputs 10 --outputs 4 -o f.sop</code>

About the tools

SIS (Sequential Interactive Synthesis, UC Berkeley) with **STAMINA** (state minimization), **NOVA** and **JEDI** (state encoding) and **espresso** (two-level minimization). The kit uses SIS 1.4 from <https://github.com/JackHack96/logic-synthesis>, built as static Linux x86-64 programs with two small fixes (source/). Licence: free for educational, research and non-profit use (bin/LICENSE_
SIS.txt).