Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme

Integrated Systems Laboratory

# TIC - TAC - TOE

Peter M. Geiser

Ambros Marzetta

Semester Project
Winter 1989/90

Assisting: Hanspeter Wachter
Responsible Professor: Prof. Dr. W. Fichtner

*Abstract*

Tic-Tac-Toe is a game played on a 3 by 3 square board. Two players set stones in turn until either one of them wins by having three of his own stones in one row or there are no more empty squares on the board, which is a draw.

This paper describes a VLSI chip playing Tic-Tac-Toe against a human opponent. To calculate its moves, it uses the Minimax algorithm with $\alpha$-$\beta$-pruning. Because of the limited length of a game, the search tree can be evaluated to the very end, the chip thus being able to determine the exact value of every possible move.

# Contents

# 1 Introduction

During the three semester course *VLSI I - III* we learn about fundamentals, design and testing of integrated circuits. During part two of this cycle we have the possibility to design one by ourselves according to our own idea, as far as this idea wasn't too complex or too easy. Obviously our idea of a *Tic-Tac-Toe chip* was alright.

## 1.1 History of Tic-Tac-Toe

*Tic-Tac-Toe* is a game developed from *Millwheel*, which is one of the oldest games of mankind. The earliest traces lead back to a burial place of the bronze age (at 2000 B.C.) near Cr Bri Chualann/Wicklow in Ireland. In old Egypt, on top of the temple of Kurna (near Theben), which was built in the XIX. dynasty under Ramses I. (1308-1306 B.C.) and Sethos I. (1305-1291 B.C.), a plan of millwheel was found amongst six other games. Millwheel was also found in antique Greece, Crete and even in the ruins of Troja. The Romans too very much liked the game. But millwheel was known even in China at 500 B.C. and 450 Years later in Sri Lanka and in many other places all over Asia and Africa. The American Indians learned about the game from the Wikings and the Spanish conquistadores.

Other common names for millwheel are *Nine Men's Morris*, in Germany *Mühle* or *Neunemal*, in Italy *Mulino*, in Holland *Driesticken* and in Spain *Tres en raya*.

## 1.2 Rules of Tic-Tac-Toe

Tic-Tac-Toe is played on 3x3 squares field. Two players alternatively set stones on these squares. The goal is – as in all millwheel-like games – to get three stones in one row. If there are no more empty squares and none of the players has three in a row, then it is a draw.

There are two variants of Tic-Tac-Toe:

**Arabian Millwheel** is played on a 3x3 squares field. Both players have three stones which they place in turn on one of the empty squares. As in all millwheel-like games, one of the players wins if he (or she) has three stones in one row. After all stones are set, the players jump. That means, they take one of the stones and place it on any other empty square.

**Achi, or African Millwheel** is like Arabian Millwheel with two exceptions: There are four stones instead of three and the stones are moved to adjacent squares only.

# 2 Algorithm

Despite the limited number of possible combinations of this game, we have chosen not to implement a big move-table, where each possible position could be looked up and thus the next move be obtained. We decided to use the wide-known MINIMAX-Algorithm (fig. 1), because it somehow generalized the problem. With more or less the same techniques, we now would be able to implement a chip that could play normal millwheel or even chess (which would lead to quite a huge chip or more probably a whole set of chips).

The name of the MINIMAX-Algorithm is based upon the idea of maximizing the score of oneself and minimizing the score of the opponent. In the following text we call the chip Max and the opponent Min. Max tries to maximize the score, which means he chooses the move which leads to the best possible position. Min, which in turn tries to negate Maxes efforts, chooses the move leading to the (for Max) worst position (which is the best position for Min). Now the turn is Maxes, which chooses ... and so on. When all possibilities are calculated, the chip makes its move. Now the player has to answer the move with his own one, over which the chip has no control. After the player moved, the whole search begins all over again. If the search tree has been memorized, the part which has been selected through the two made (half-) moves can be copied and being continued. This optimization needs much memory, depending on the depth of the search. Memory and the small search tree were the reason for us not to choose this optimization method.

So far, the only thing done is choosing the right move. Now, which is the right move or how do we find out which is the best (or worst) position after each move? In Tic-Tac-Toe, that is no problem, because of the limited number of possible positions. So we calculate the whole search tree at the very end. The end is either Win, Draw or Loose (always seen from the chip). So we don't need a position evaluation routine. That couldn't be done for chess. In chess we would have to cut the search tree after about 12 to 16 moves and then determine the value of this newly gained position. The success of a chess program very much depends on how good it can evaluate the value of a certain position.

Search trees can become very big, even with limited search depth. So we would like to cut unnecessary parts of the tree. For instance, if one part of the tree leads to win, we are no longer interested in other possibilities, since we achieved our goal. This leads to the so called $\alpha$-$\beta$-pruning (John McCarthy, 1956). He found that $\alpha$-$\beta$-pruning allows an average 33% deeper look-ahead.

For $\alpha$-$\beta$-pruning, we have to use two boundaries, $\alpha$ and $\beta$, respectively. $\alpha$ is the lower limit for the value that Max finally can have, $\beta$ is the highest value of Min.

First of all, we implemented the game in software. Since we were working on a Sun 3/60 under the Unix operating system, the logical choice was a C program, which is to be found in appendix F.

Inputs : - Tic-Tac-Toe board
- Whose turn is it?

Outputs: - Value of this position (as seen from the current player)
- best possible move

Algorithm:

```
IF three stones in one row THEN
   value := +1
ELSIF board full THEN
   value := 0
ELSE
   bestr := -1
   FOR each possible move i DO
      r := - value of resulting position seen by opponent
      IF r = +1 THEN
         RETURN ( i is best possible move )
      ELSIF r > bestr THEN
         m := i
         bestr := r
      END
   END
   m is best possible move
END
```

Figure 1: MiniMax-Algorithm

# 3 Architecture

Based on the algorithm we have chosen the architecture of the chip. The whole thing is partitioned into five Blocks (fig. 2). The computation is carried out in three of them (`board`, `move` and `value`), being controlled by a finite state machine (`fsm`), and – because the algorithm consists of a recursive procedure – data is pushed onto a stack (`stack`). `Move` is the part which generates and remembers moves, `value` does everything concerning the evaluation of moves and `board` stores the nine place board.

## 3.1 Board

`Board` contains two registers. The first register (`p`) corresponds to the parameter `p` in the algorithm. A value of zero signifies that the chip has to move and a one that the user has to move. This register is initialized to zero during reset because the first invocation of the subroutine computes a move for the chip. At every recursive call of the subroutine, the value of `p` is inverted. So we need not push its value onto the stack like the other registers, but can change it back by the same inverter when we return from the subroutine.

The content of the other register (`board`) describes the state of the board. There are nine places (numbered from 0 to 8). Each of them can be empty, or contain a white or a black stone. Thus we need 2 bit per place; that makes 18 bit for the whole board. An addressing logic allows to invert one single bit of this register per cycle. The four most significant bits of the address come from the move block (`index`). The least significant bit is the output of the `p` register. The `toggleN` input decides whether the addressed bit is inverted or not.

At the output of the 18-bit register there is a lot of combinatorial logic. One circuit (`win`) decides whether there are three stones of the same color in a row, i. e. whether one of the two players wins the game. Another circuit (`full`) decides whether the board is filled with 9 stones. In this case, the game would be finished. If the player presses a key, a third circuit is able to check whether this is a legal move. If it isn't legal, the key will simply be ignored.

The fourth circuit (`findfree`) gets the number of a place on the board (`i`) and returns the number of the next free place (`nextfree`). For instance, if it gets the number 2, and places 3, 4 and 5 are occupied by white or black stones, the circuit will return 6. This corresponds to the statement in the algorithm which says that every possible move has to be tried one after the other. Of course the circuit has an initialization flag (`usei`) making it choose the first free place on the board. There is also an output flag (`lasti`) indicating whether there are still free places on the board or we are at the end of the loop. This circuit is the one with the greatest propagation delay on the chip – it limits the clock frequency. This is due to the fact that it contains a signal which is propagated similarly to the carry chain in a 9 bit ripple carry adder.

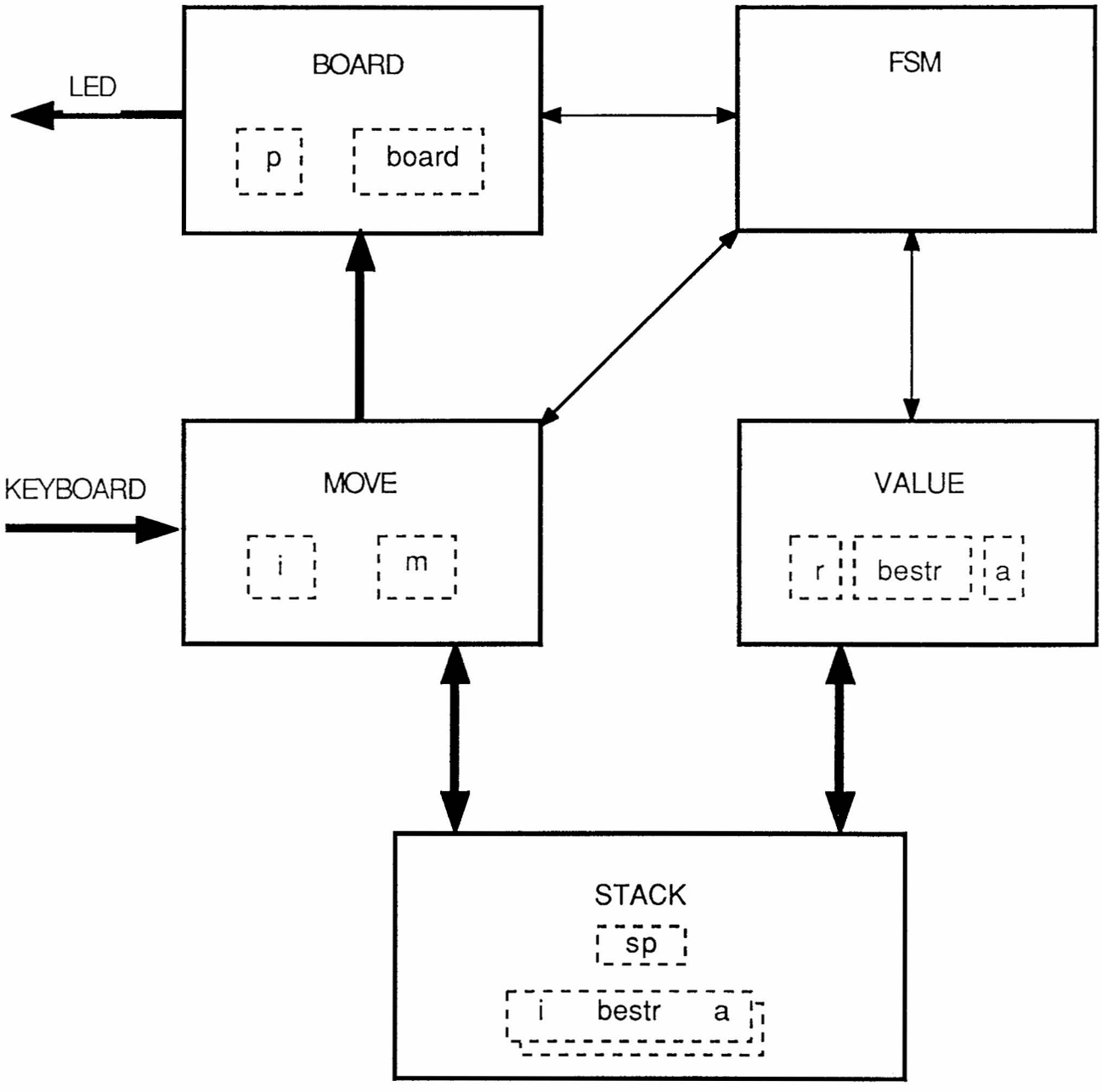



Figure 2: Architecture of the chip

## 3.2 Move

The move block contains the variables `i` and `m` from the algorithm, a random number generator and a multiplexer. `I` is the move being tried. It is a number from 0 to 8, which fits into a four bit register. It is set to a free place on the board by the signal `nexti`. If a move is better than all its predecessors, the `assignmove` signal assigns the value of `i` to `m`.

For the first move in a game (when the board is empty) we can choose any place we want, we can always force a draw. But the chip must not make the same starting move in every game – this would be boring. The first move of the chip is selected by a random number generator. It generates three-bit random numbers simply by starting at an undefined value and counting endlessly.

The multiplexer selects the index of the bit to be inverted in the board register from four values: the random number, `i`, `m` and the key pressed by the user. It is controlled by the two-bit signal `choose`.

## 3.3 Value

`Value` contains the variables for static evaluation of the current position. In `r` we store the value of the current move, gained by a win-position or a full board. The value of the best move possible in a certain position is stored in `bestr`. The signal `rgbestr` indicates whether or not `r` is greater than `bestr`. If `r` is greater than or equal to `a`, which is signaled by `rgea`, the calculation of a possible move at this position finishes.

`A` is changed if one of `call` and `pop` is high. In the first case, `a` becomes the value of `bestr`, in the latter case, `a` becomes the value of a pushed `a`, which was pushed on the stack earlier.

In case of `pop`, `bestr` behaves just the same as a in restoring the last pushed value. If `call` or `init` is high, `bestr` is initialized to one, `remember` leads to a zero.

There are four signals to control `r`, plus one input. `clearr` sets `r` to 0, `sno` sets `r` to -1 and `negr` negates `r`. `retbest` assigns `r` the value of `bestr`.

## 3.4 Stack

The stack is organized as an 8 by 6 bit register. The six bits contain copies of `i` (4bits), `a` and `bestr`. The depth of 8 is given by the maximum of look-ahead moves to store.

## 3.5 The controller

The controller is implemented by a finite state machine. It has eight states. A reset puts it into the state *justreset*. There it sees three possibilities. Either the user begins, the chip begins or the chip is being tested. It chooses the third possibility if there is already a key pressed. Otherwise it selects one of the first two possibilities according to the switch `userbegins`.

In the state *waiting* it waits for the user to press a key and to do his move. When the user finally presses one and it designates a place which is not yet occupied, the machine sets a stone of the user's color on the place and goes to state *userjustmoved*.

In the state *userjustmoved* it decides whether the board is full or not. If it is full, the game has ended with a draw and the machine goes to state *finished*. It needn't test whether the user has three in a row because it is impossible for the user to win. If the game is not yet finished, the chip sets `i` to the number of the first free place and goes into state *setboardi*. We are now in the middle of the Minimax algorithm.

In the state *setboardi* the chip only places a stone for player `p` onto the board and goes to state *rating*.

In the state *rating* the chip's action depends on whether we are in the first (bottom) invocation of the recursive procedure or in an inner call. If we are at the bottom and have three in a row or nine stones on the board, we go straight to state *finished*. If we are not at the bottom of the recursion, but have three in a row or nine stones on the board, we know the value of the move and can return from the procedure (state *returned*). In all other cases we push the variables onto the stack, make the recursive call and go to state *setboardi*.

When the chip returns from a call of the procedure (state *returned*), it has nine possibilities depending on the values of `r`, `bestr`, `a`, whether we are at the bottom of the recursion (`ground`), and whether we are at the end of the loop (`lasti`). In most of the cases, we must undo the move we did in state *setboardi*. If we are at the end of the loop and must do the best move we have found, the machine goes to state *doingbestmove*.

In the state *doingbestmove*, the machine makes the move stored in `m` and then waits for the user's move in state *waiting*.

When the game is *finished*, the machine waits forever in the state with this name.

A formal definition of the finite state machine can be found in appendix C.

# 4 Interface

There are 49 pins used as interface between the chip and the outside world.

### Inputs

`keyboardN0 - keyboardN8` These (active low) signals tell the chip which keys are pressed. When at the end of a reset a key is pressed, the user can input a random position on the board. To enter the chip's stones, `piN` has to be high.

`userbegins` tells the chip whether the user begins (h) or not (l).

`piN` normally is low. If put to high, the user can set stones for the chip on the board.

`cp` is the clock.

`resetN` resets the chip.

`scan` puts the chip into test-mode.

`scanin` is the scan-path input.

### Outputs

`led0 - led17` show the contents of the board. Since each square can have one of three states, it is necessary to have two outputs per square.

The possible states are:

| | |
|---|---|
| empty | 00 |
| chip | 01 |
| user | 10 |

with the higher numbered bit left and the lower bit right.

`copyboard` is used to indicate whether the board contains valid information or the pre-calculated moves of the chip.

`prompt` asks the player to enter a move.

`fin` indicates whether the game is finished or still in progress.

`win` is high if the chip wins, low in case of a draw.

`i0 - i3` show the content of the register `i`.

`sp0 - sp2` show the stackpointer.

`scanout` is the output of the scan-path.

# 5 Experiences with VTI

VTI is a powerful CAD tool for design and testing of integrated circuits, but there are some things which aren't written in the manuals. That's why we write down some hints to future users of this software here.

If you want to edit an icon, make a draft version using VTIIconEdit. For exact adjustment of objects, it is much quicker to use a text editor than to move them pixel by pixel in the icon editor.

It is possible to edit cells with a text editor outside VTI even during a VTI session. But there is one thing to take care of. Having saved the cell from the editor, you must force VTI to reread the cell from disk. In the VTIshell environment this is possible with the `manager restore` command. In the graphic environment it is possible with the `cell cache` entry in the background menu.

If you change the name of a cell outside VTI, it doesn't suffice to change the file name. The cell is required to contain its own name in the first line of the file.

After a crash of VTI, it is sometimes necessary to manually remove the lock file.

In order to simulate a chip containing a datapath, you need two different switch cells: One for simulation of the netlist derived from the schematic and one for the post-layout-simulation (fig. 3).

| | On/Off | Src Typ | Src Name | Src Pcl | Inst | Dest Typ | Dest Name | Dest Pcl |
|---|---|---|---|---|---|---|---|---|
| Pre-layout | on | mde | mdp | movel | * | hns | movel_L | |
| Post-layout | on | lph | movel | | * | hns | movel_L | |

Figure 3: Switch cells

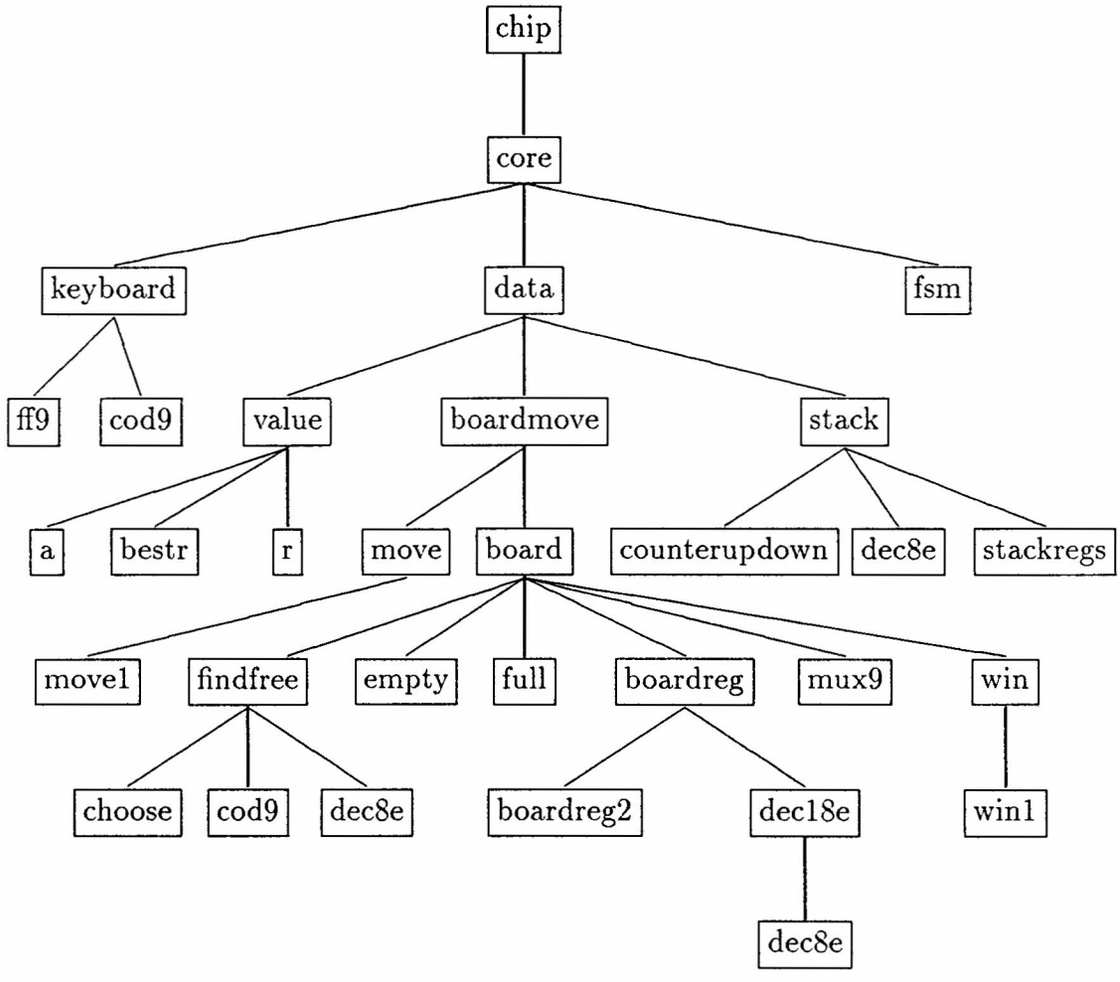
# 6 Conclusions

When we began our design project, we defined the following aims: We wanted to design a system being able to play a game against a human opponent or another chip. Besides the chip itself, the system should need as little additional components as possible. We have fully attained these goals. The only additional components needed are a clock generator, 18 LEDs and a few buttons. The chip will work at a clock frequency of 6MHz, and compute its move in at most one millisecond.

During the project we learnt a lot about digital circuits in general and about VLSI specifically. We are very satisfied about having had the possibility to plan a chip of such a size which will really be produced. We also have learnt to deal with a huge software package (VTI). Sometimes the problems coming from the software are much greater than the original problem to be solved using the software.

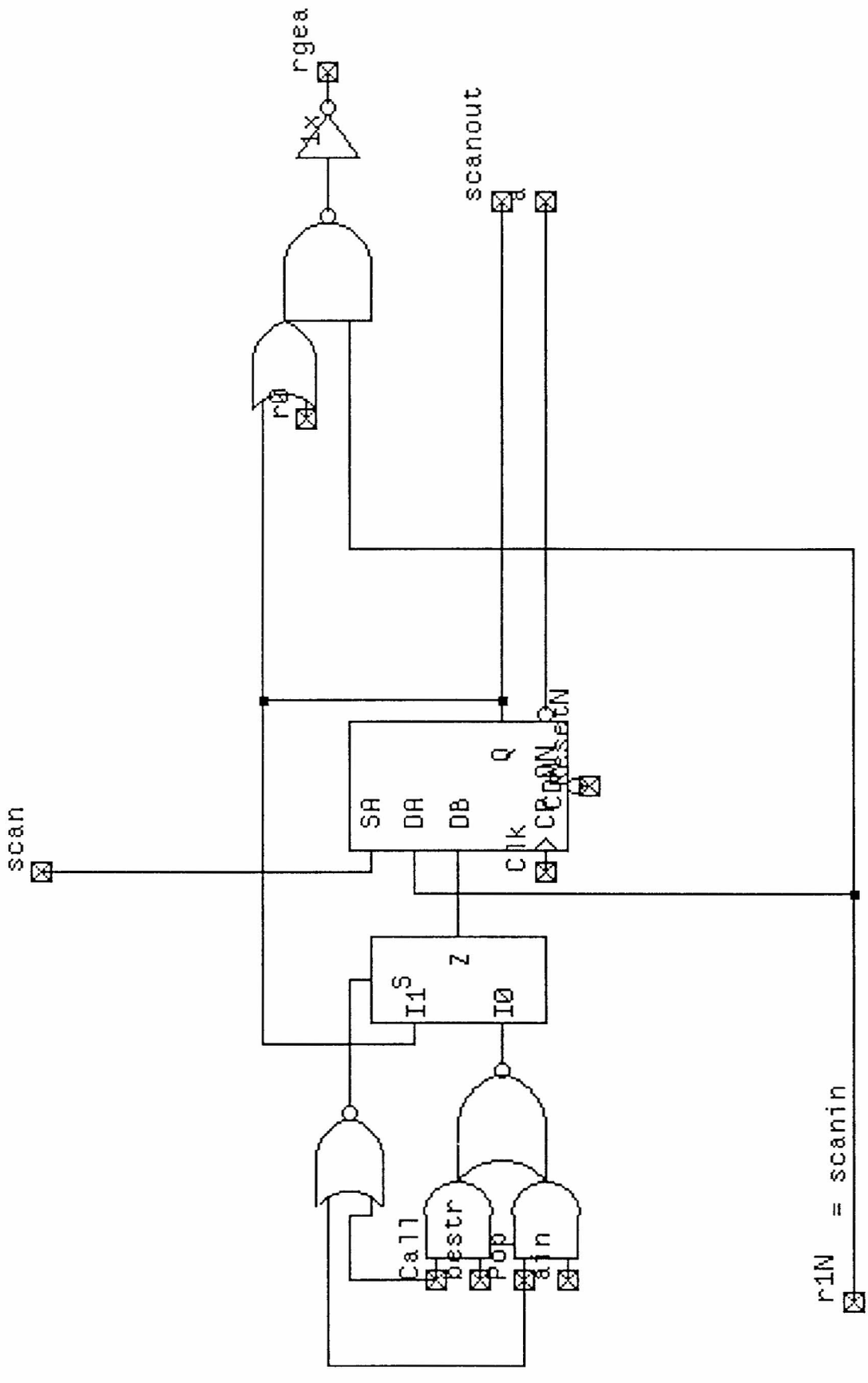
We have really enjoyed the working conditions and that we were free to implement our own ideas and to work in our own way. All assistants we asked anything helped us willingly. To all them we express our gratitude here.

# A Cell Hierarchy

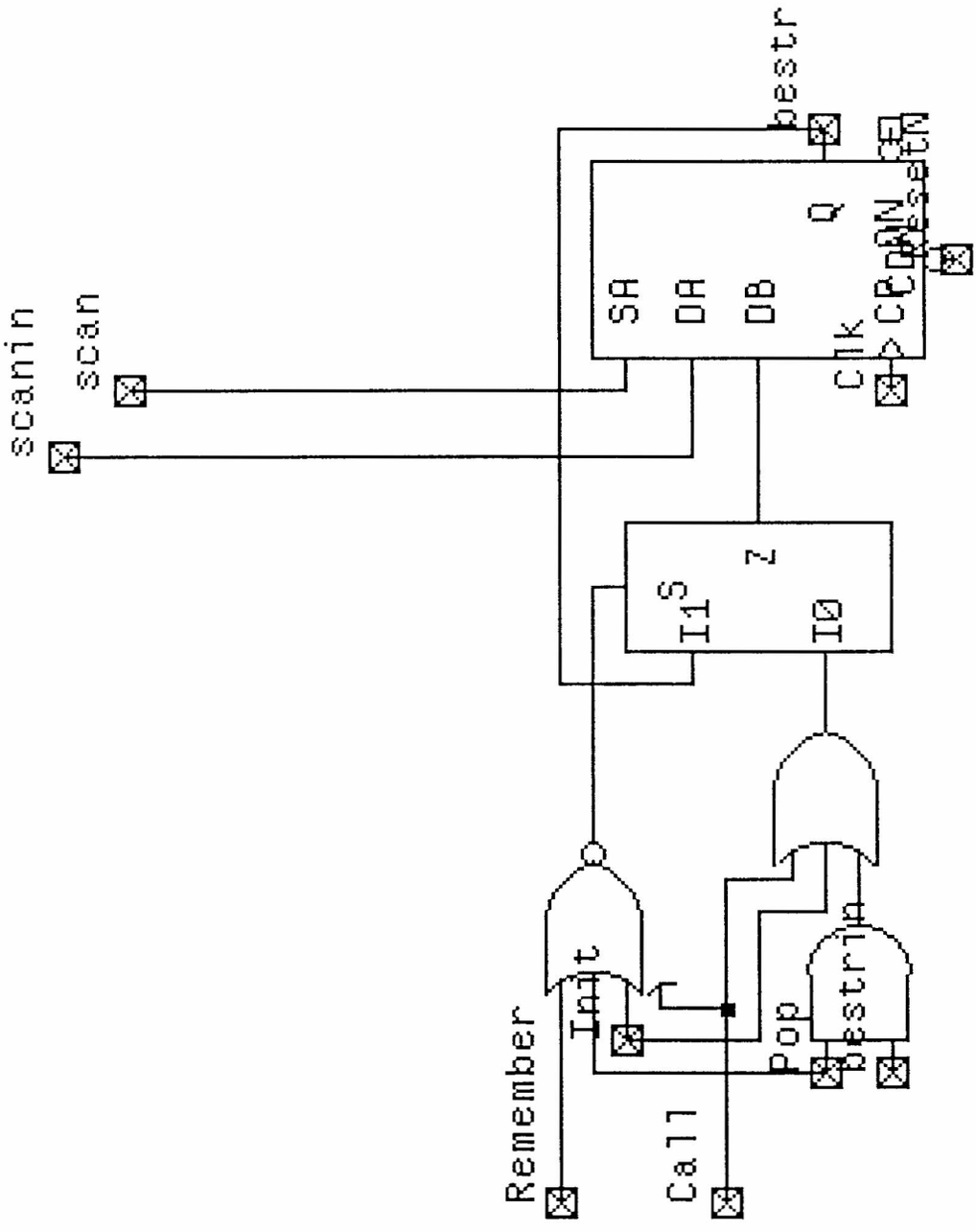


# B Schematics

a


rgea
scanout
rq
scan
SA
DA
DB
Q
Clk
I1
S
Z
I0
Call
bestr
Pop
ain
r1N = scanin

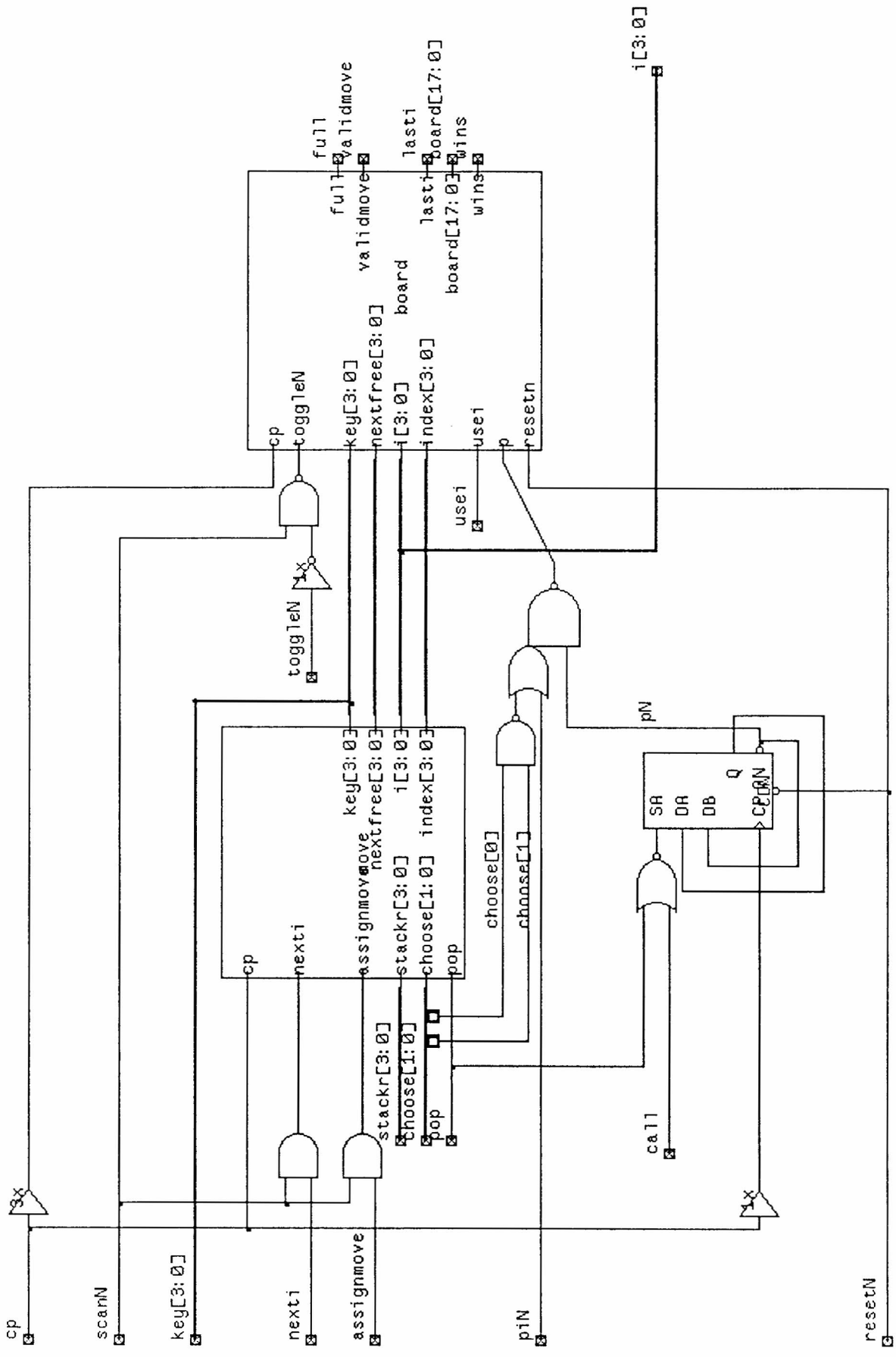
bestr



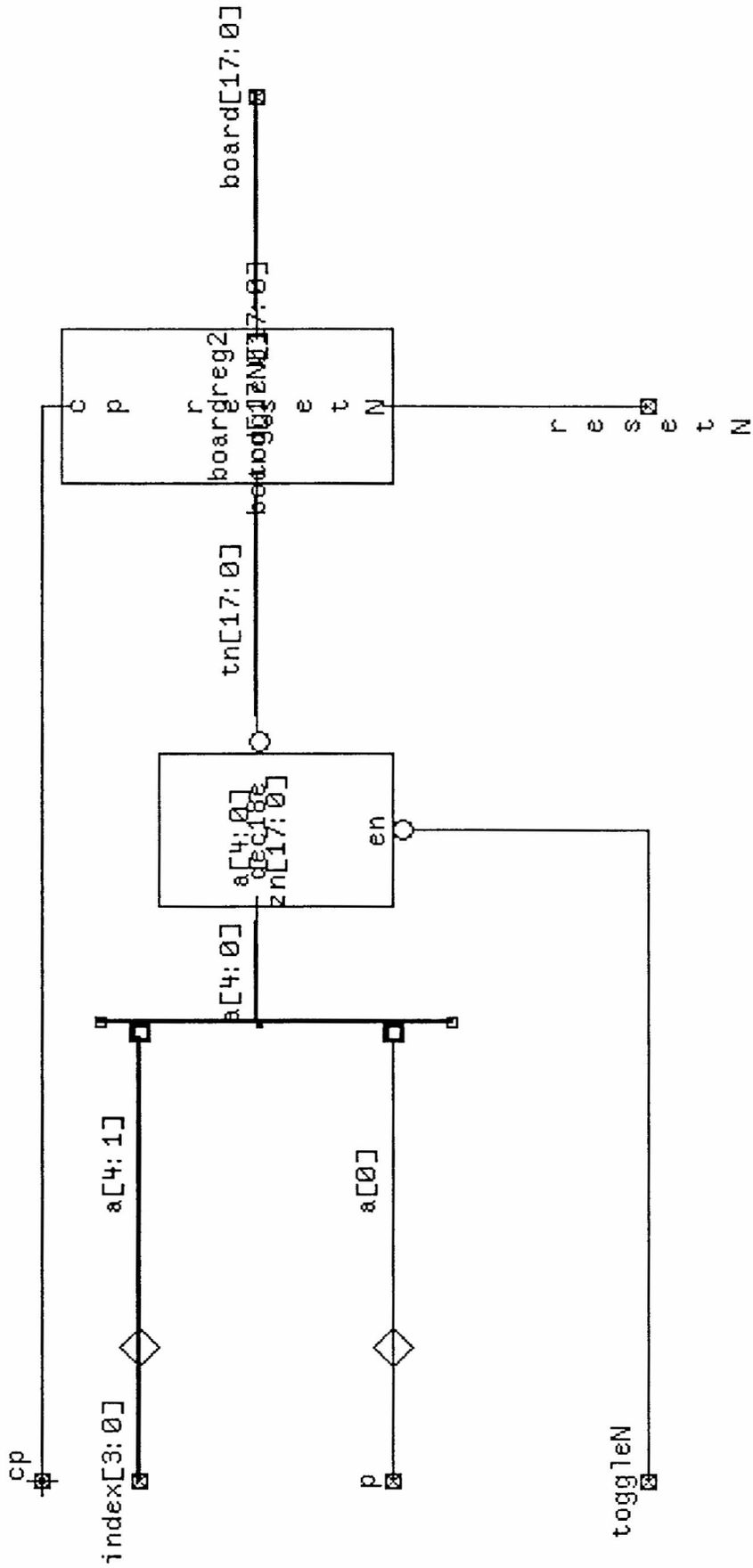
board


full
validmove
nextfree[3:0]
lasti
board[17:0]
wins
full
empty[8:0]
I[8:0]
mux9
S[3:0]
Z
findfree
usei
lasti
win
b[17:0]
wins
empty[8:0]
e[8:0]
empty
b[17:0]
key[3:0]
usei
cp
index[3:0]
p
boardreg
s
toggleN
t
N
resetn

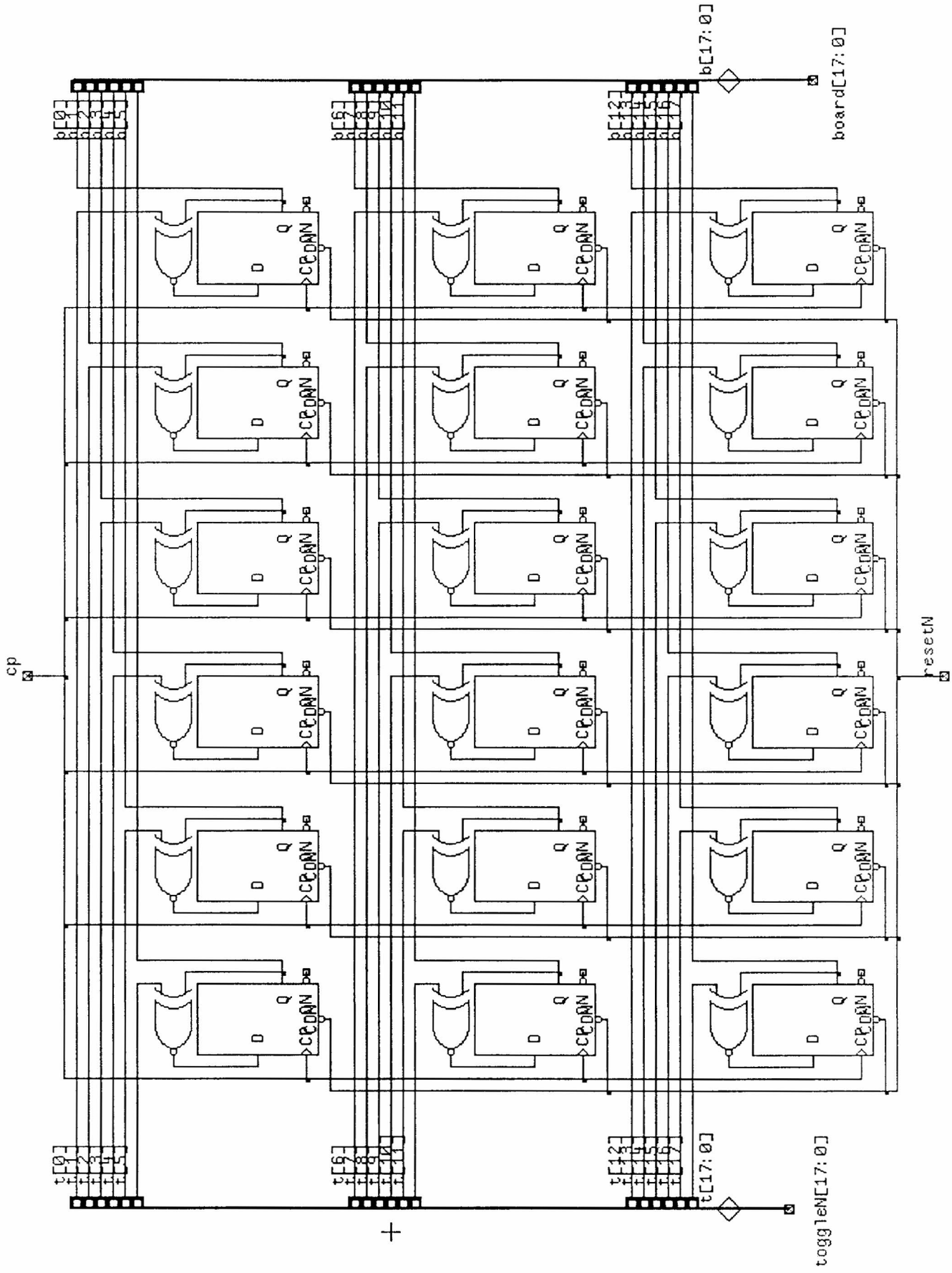
boardmove


cp
scanN
key[3:0]
nexti
assignmove
piN
resetN
call
pN
choose[0]
choose[1]
stackr[3:0]
choose[1:0]
pop
i[3:0]
board[17:0]
full
validmove
lasti
wins
board
nextfree[3:0]
index[3:0]
usei
resetn
toggleN
SA
DA
DB
Q
CP
CPN
3x
1x

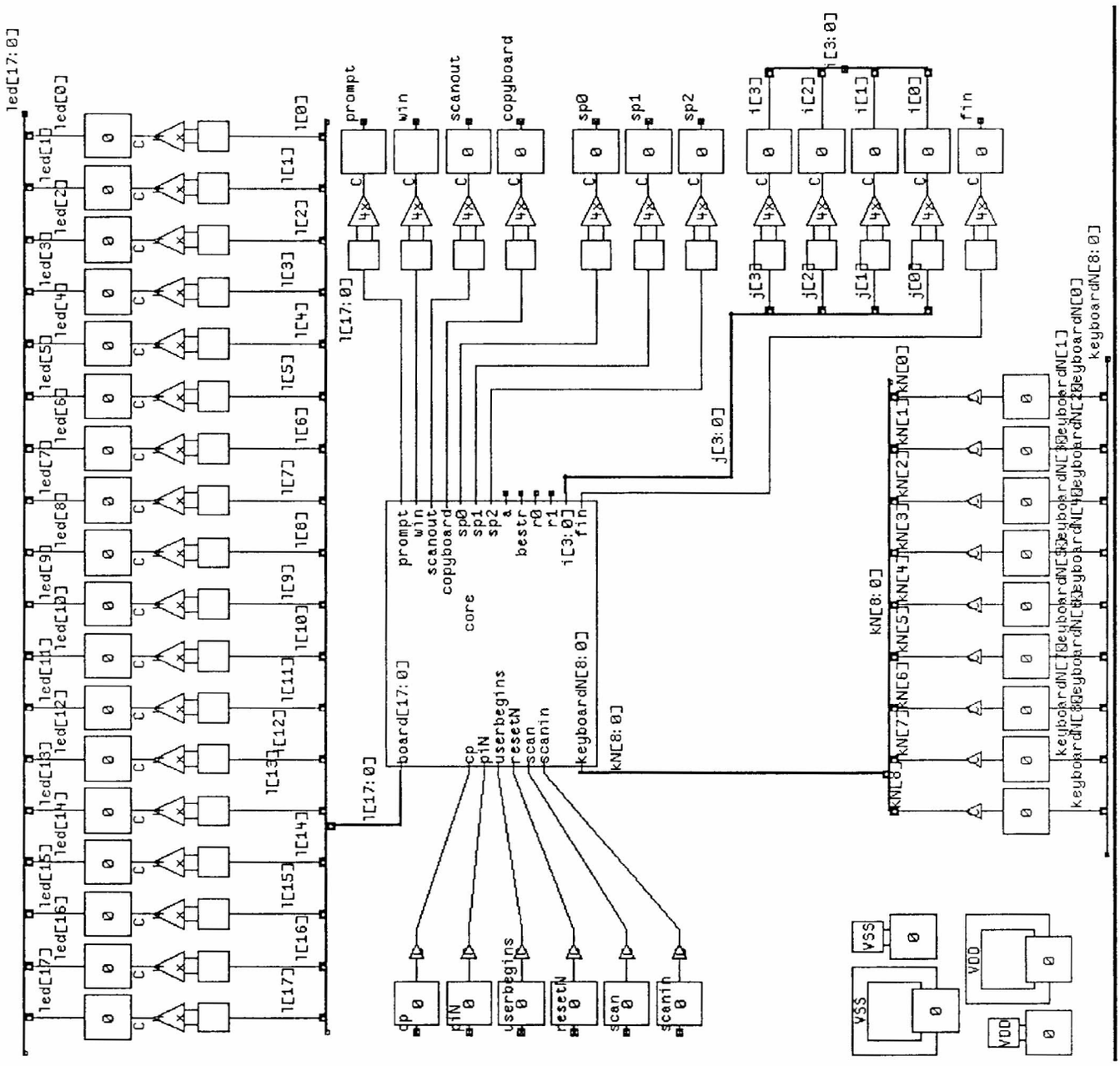
boardreg



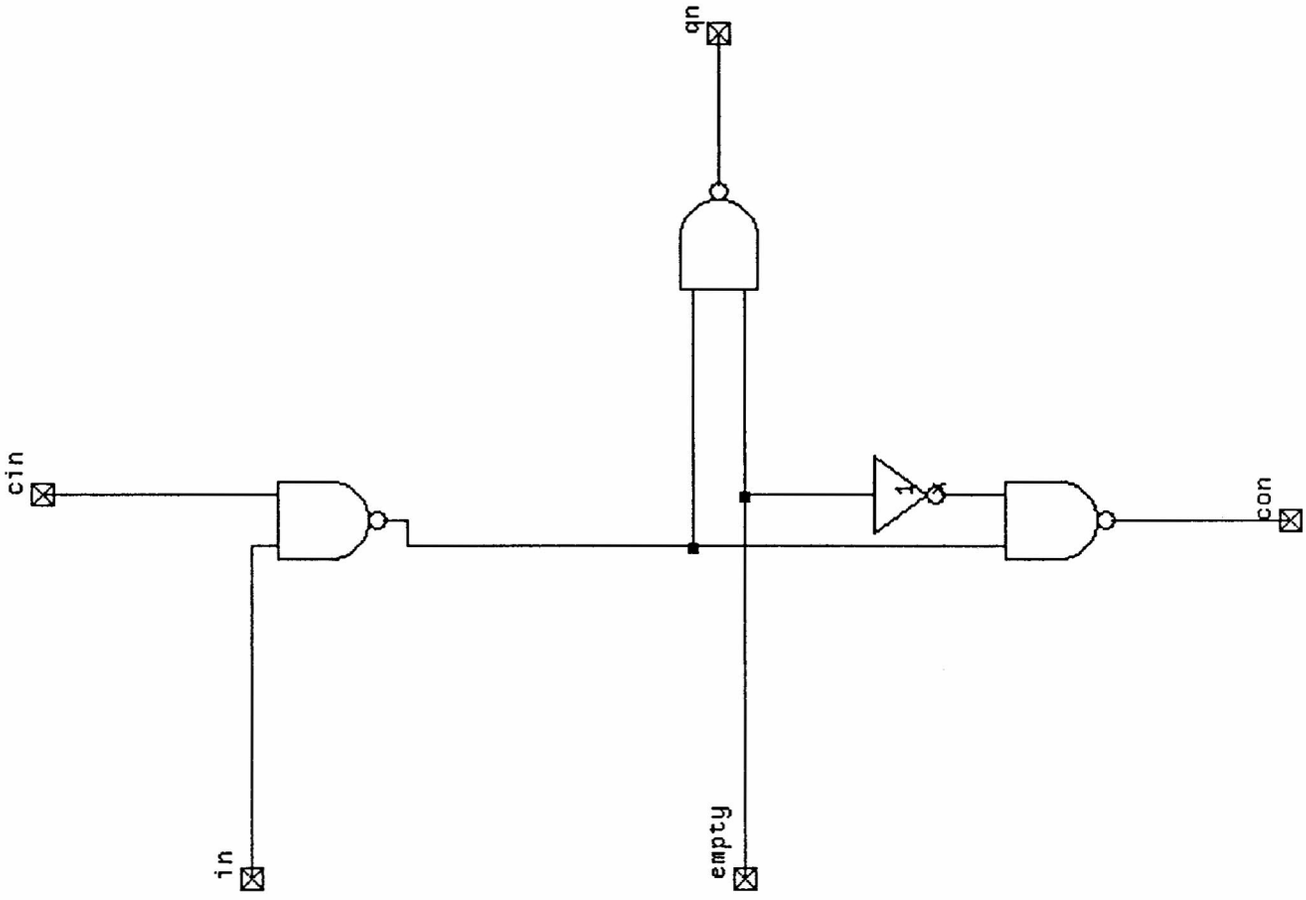
boardreg2



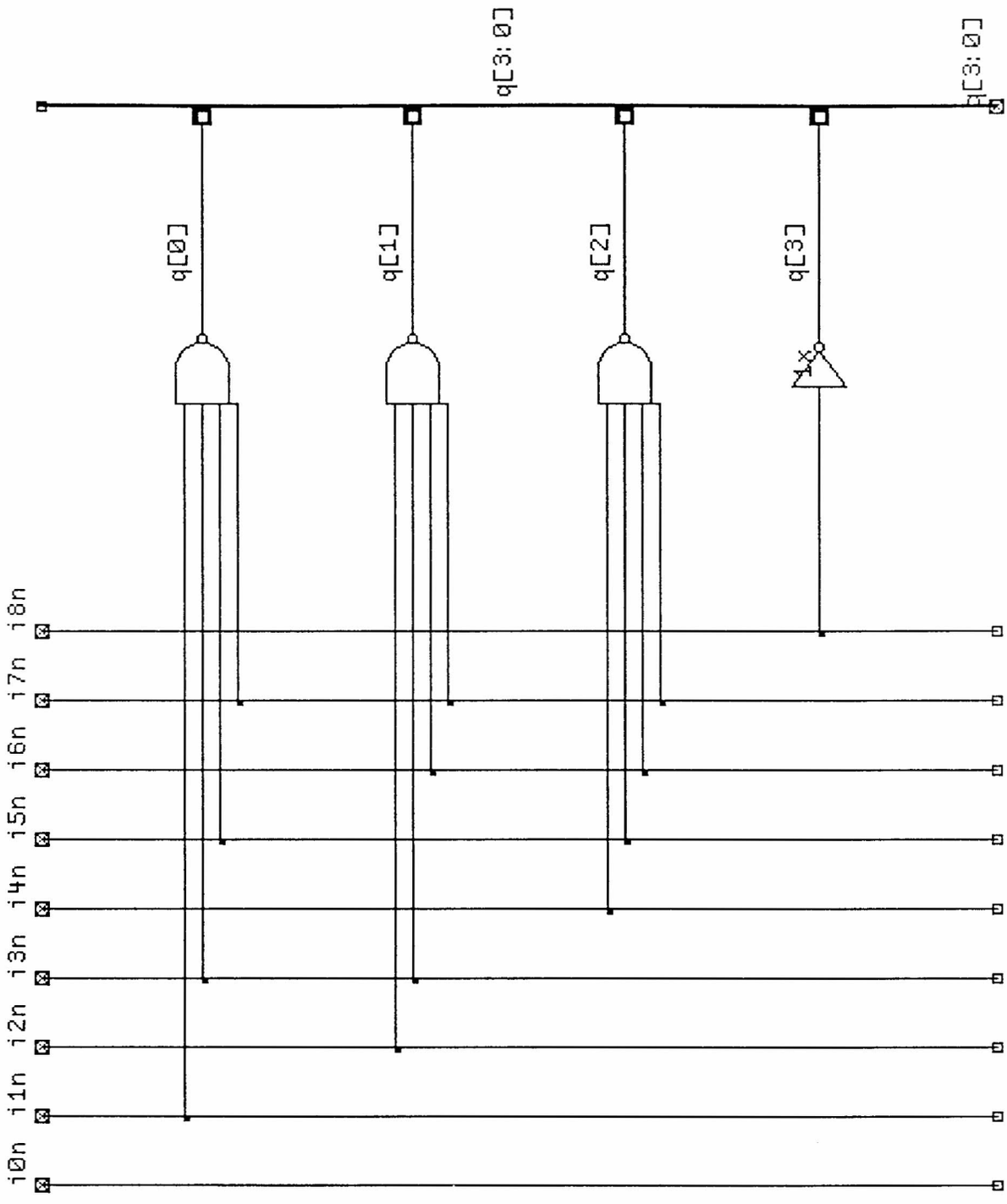
chip

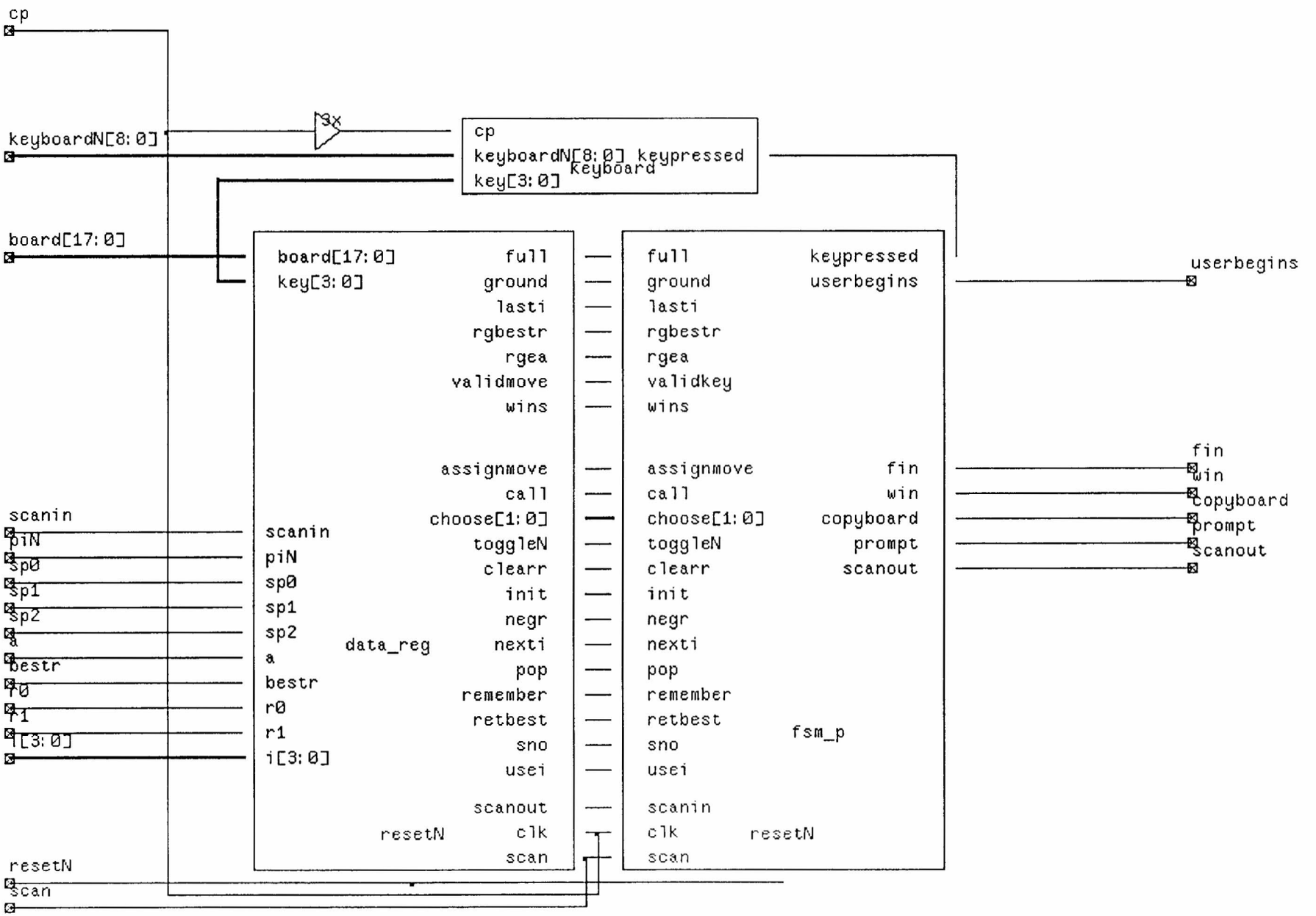

led[17:0]
led[0]
led[1]
led[2]
led[3]
led[4]
led[5]
led[6]
led[7]
led[8]
led[9]
led[10]
led[11]
led[12]
led[13]
led[14]
led[15]
led[16]
led[17]
l[0]
l[1]
l[2]
l[3]
l[4]
l[5]
l[6]
l[7]
l[8]
l[9]
l[10]
l[11]
l[12]
l[13]
l[14]
l[15]
l[16]
l[17]
prompt
win
scanout
copyboard
sp0
sp1
sp2
i[3]
i[2]
i[1]
i[0]
fin
i[3:0]
l[17:0]
j[3:0]
j[3]
j[2]
j[1]
j[0]
core
board[17:0]
kN[8:0]
keyboardN[8:0]
cp
pIN
userbegins
resetN
scan
scanin
bestr
r0
r1
a
sp
kN[0]
kN[1]
kN[2]
kN[3]
kN[4]
kN[5]
kN[6]
kN[7]
kN[8]
keyboardN[0]
keyboardN[1]
keyboardN[2]
keyboardN[3]
keyboardN[4]
keyboardN[5]
keyboardN[6]
keyboardN[7]
keyboardN[8]
VSS
VDD

choose

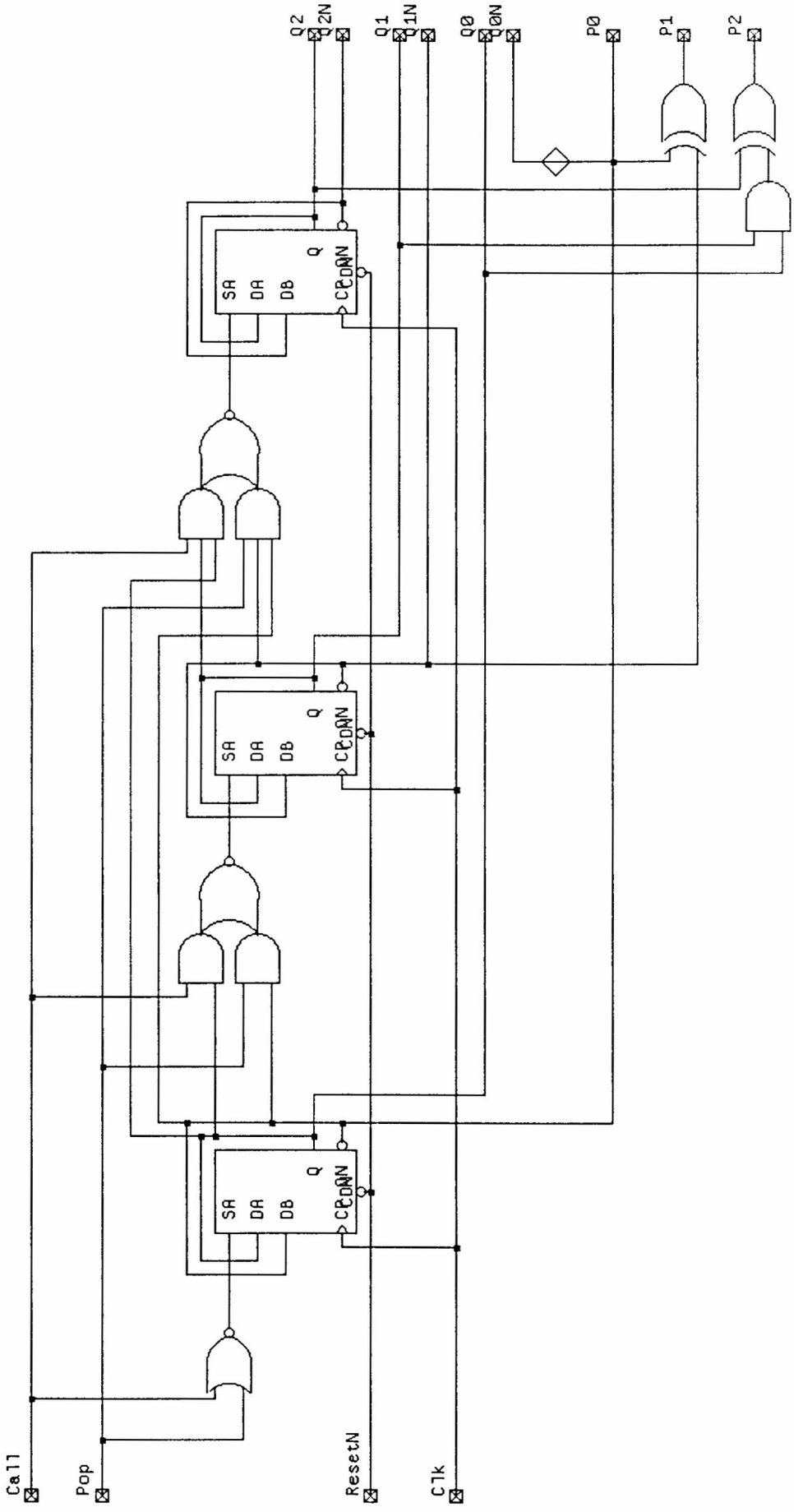


cod9

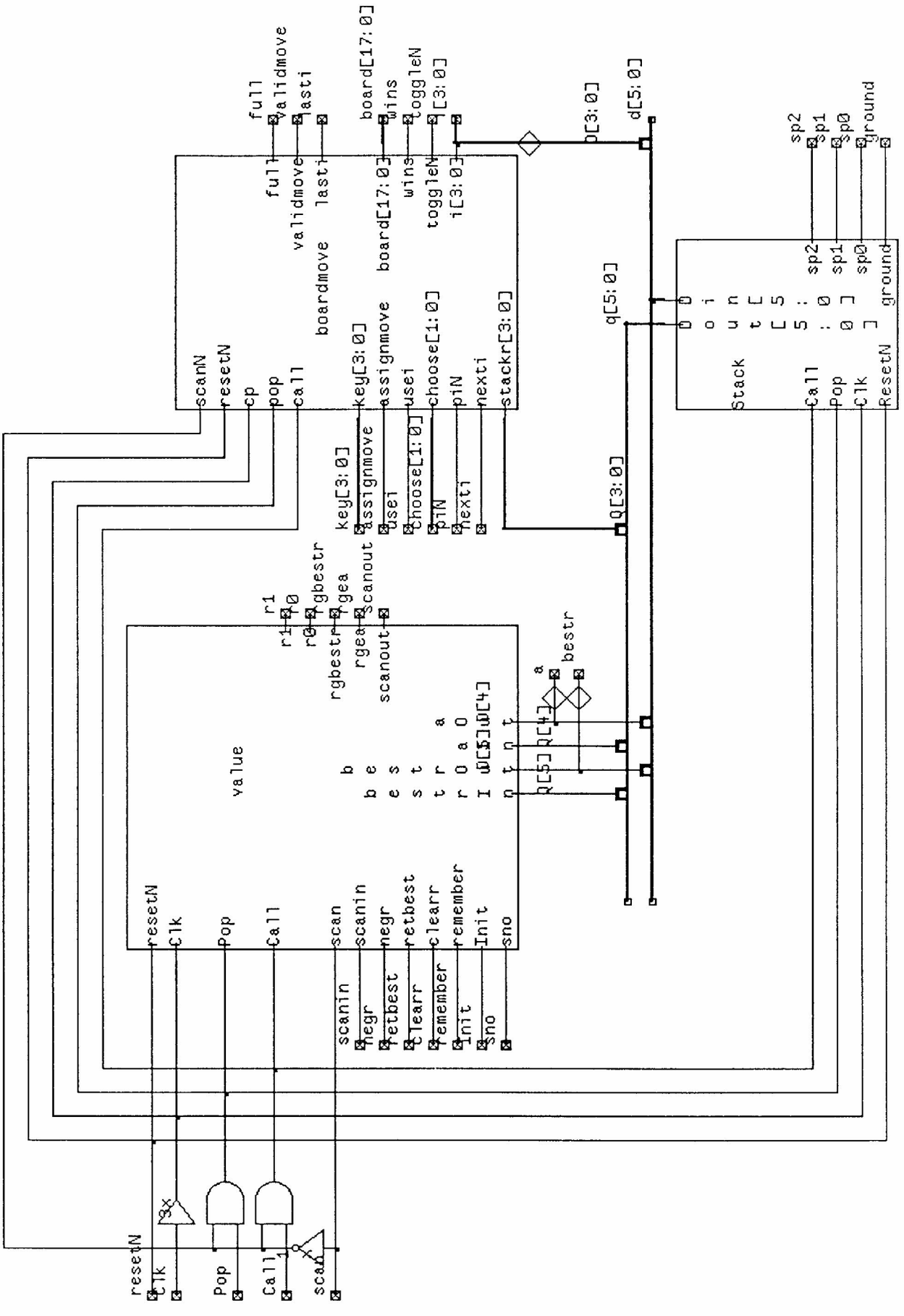



cp
keyboardN[8:0]
3x
cp
keyboardN[8:0] keypressed
key[3:0]
keyboard
board[17:0]
board[17:0]
key[3:0]
full
ground
lasti
rgbestr
rgea
validmove
wins
assignmove
call
choose[1:0]
toggleN
clearr
init
negr
nexti
pop
remember
retbest
sno
usei
scanout
clk
scan
resetN
data_reg
full
ground
lasti
rgbestr
rgea
validkey
wins
assignmove
call
choose[1:0]
toggleN
clearr
init
negr
nexti
pop
remember
retbest
sno
usei
scanin
clk
scan
resetN
fsm_p
keypressed
userbegins
fin
win
copyboard
prompt
scanout
userbegins
fin
win
copyboard
prompt
scanout
scanin
piN
sp0
sp1
sp2
a
bestr
r0
r1
i[3:0]
resetN
scan
core

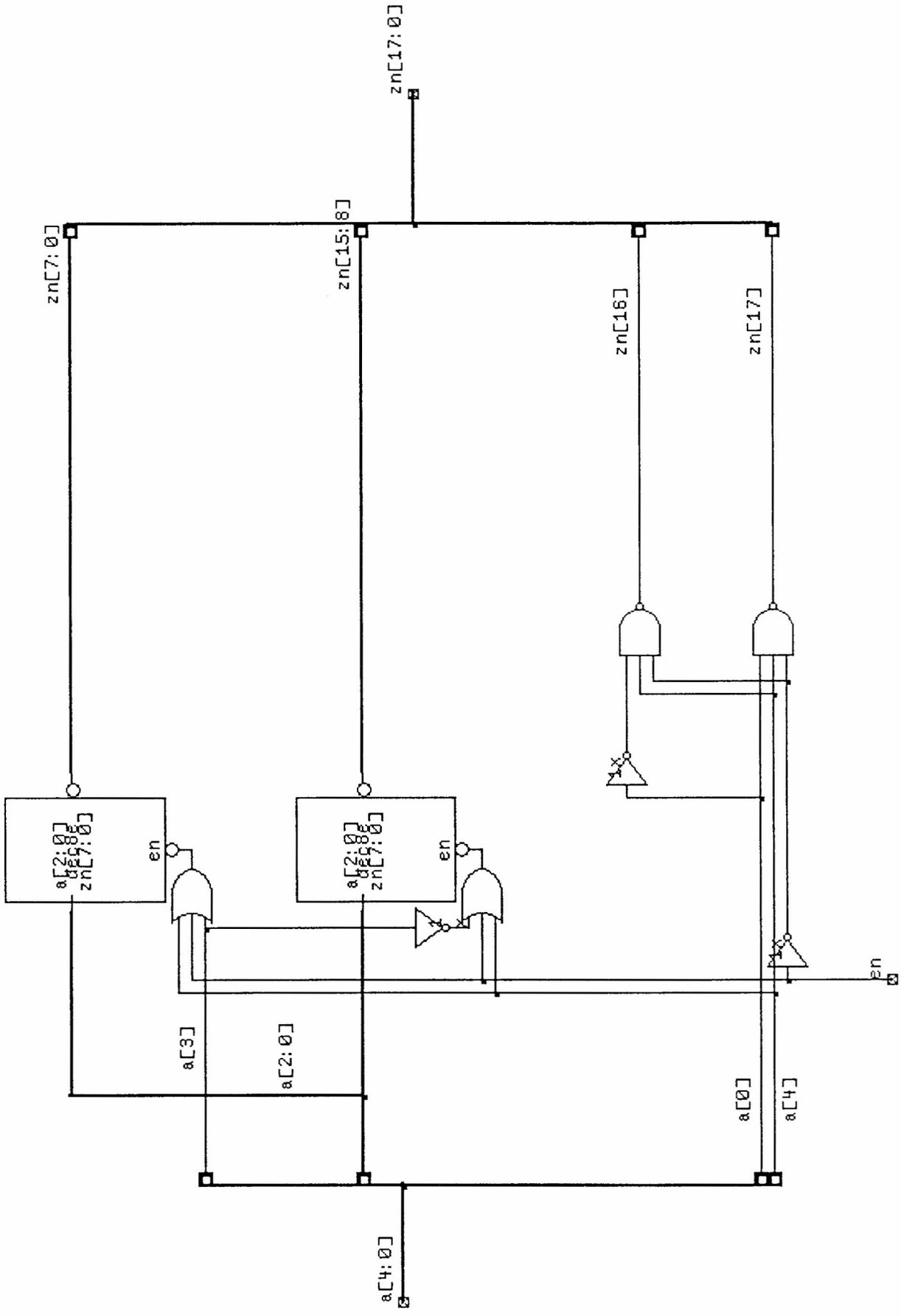
counter



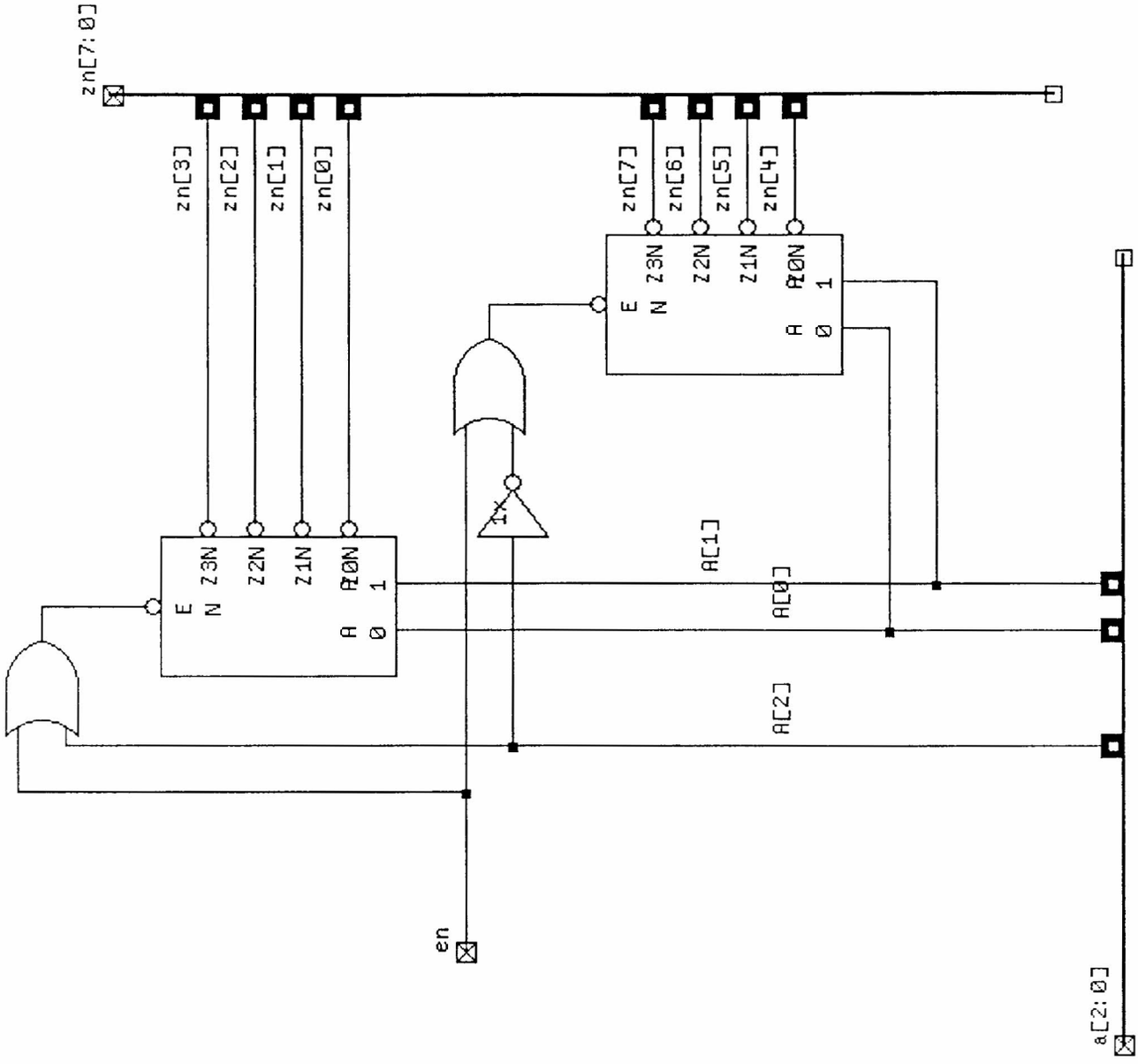
data


resetN
Clk
Pop
Call
scan
scanin
negr
retbest
clearr
remember
Init
sno
value
rgbestr
rgea
scanout
key[3:0]
assignmove
usei
choose[1:0]
piN
nexti
stackr[3:0]
boardmove
validmove
lasti
full
board[17:0]
wins
toggleN
i[3:0]
p[3:0]
d[5:0]
q[5:0]
Q[3:0]
Q[5]
Q[4]
bestr
Stack
sp2
sp1
sp0
ground

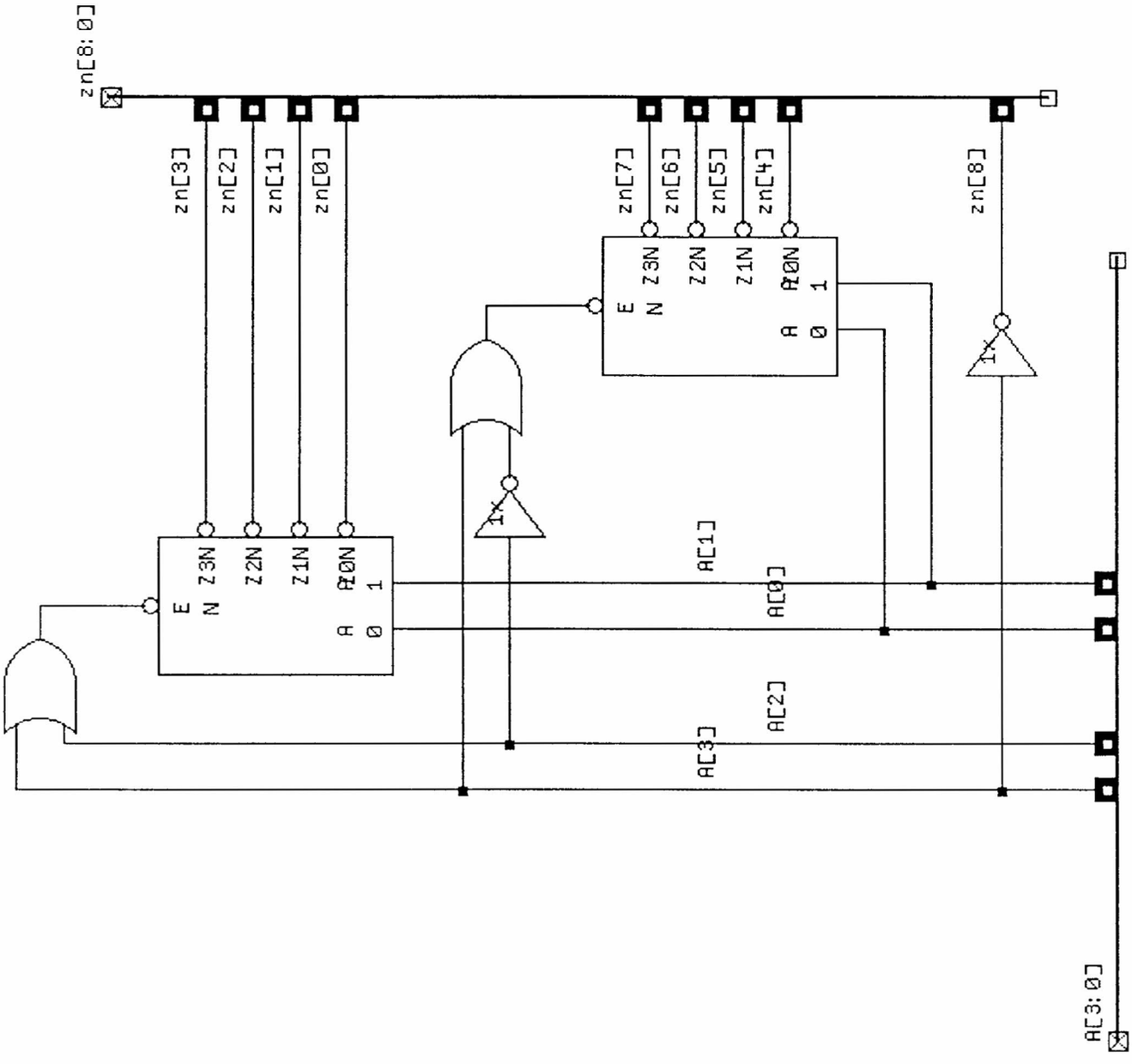
dec18e


zn[17:0]
zn[7:0]
zn[15:8]
zn[16]
zn[17]
a[2:0]
zn[7:0]
en
a[2:0]
zn[7:0]
en
en
a[3]
a[2:0]
a[0]
a[4]
a[4:0]

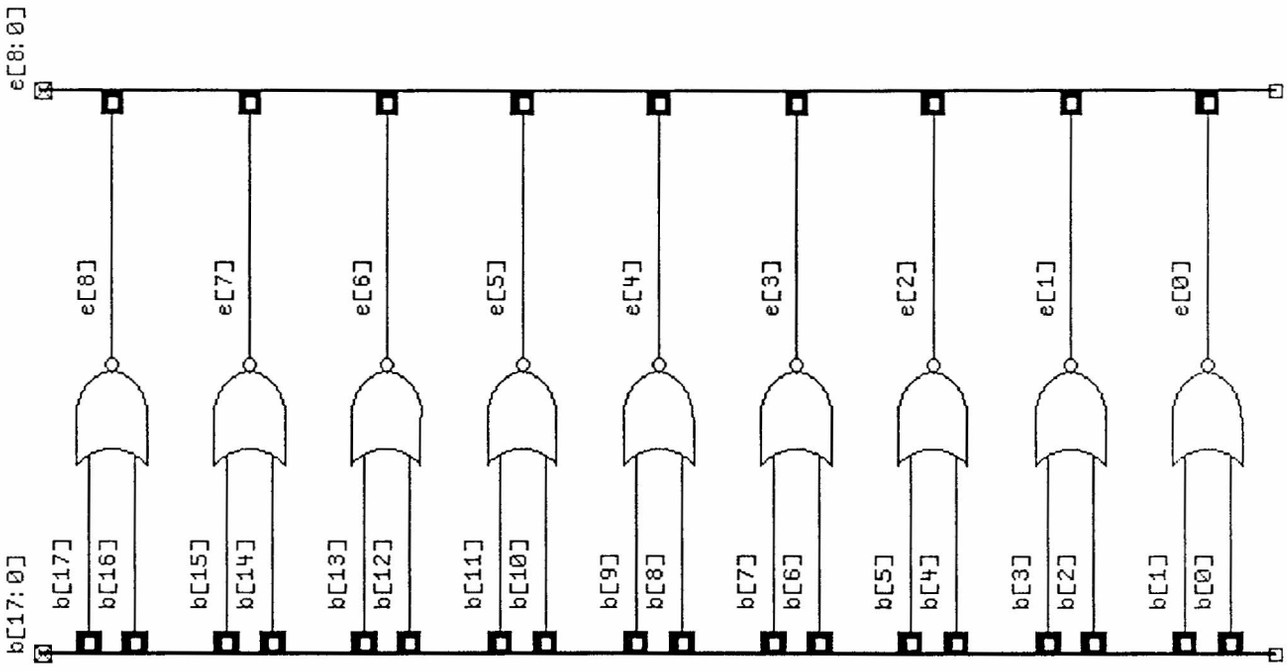
dec8e


zn[7:0]
zn[3]
zn[2]
zn[1]
zn[0]
zn[7]
zn[6]
zn[5]
zn[4]
E
N
Z3N
Z2N
Z1N
Z0N
A
0
1
A[1]
A[0]
A[2]
en
a[2:0]

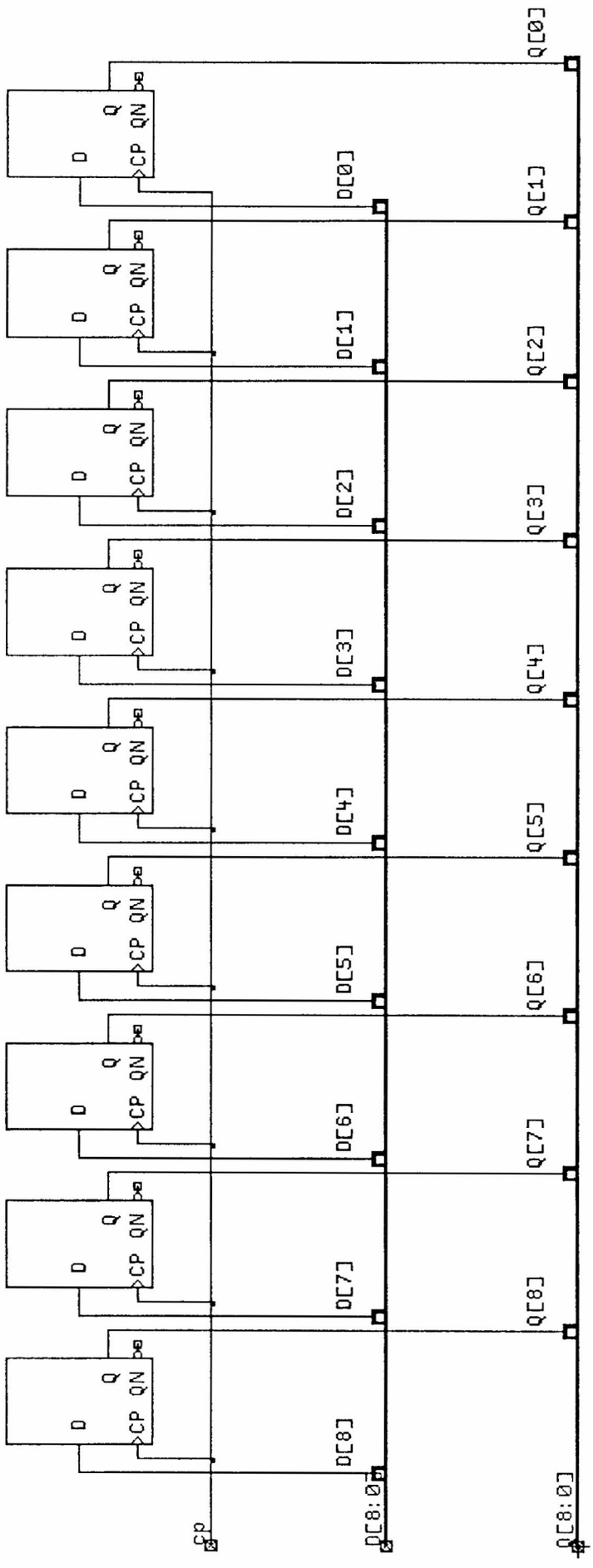
dec9


zn[8:0]
zn[3]
zn[2]
zn[1]
zn[0]
zn[7]
zn[6]
zn[5]
zn[4]
zn[8]
E
N
Z3N
Z2N
Z1N
Z0N
A
0
1
1
A[1]
A[0]
A[2]
A[3]
A[3:0]

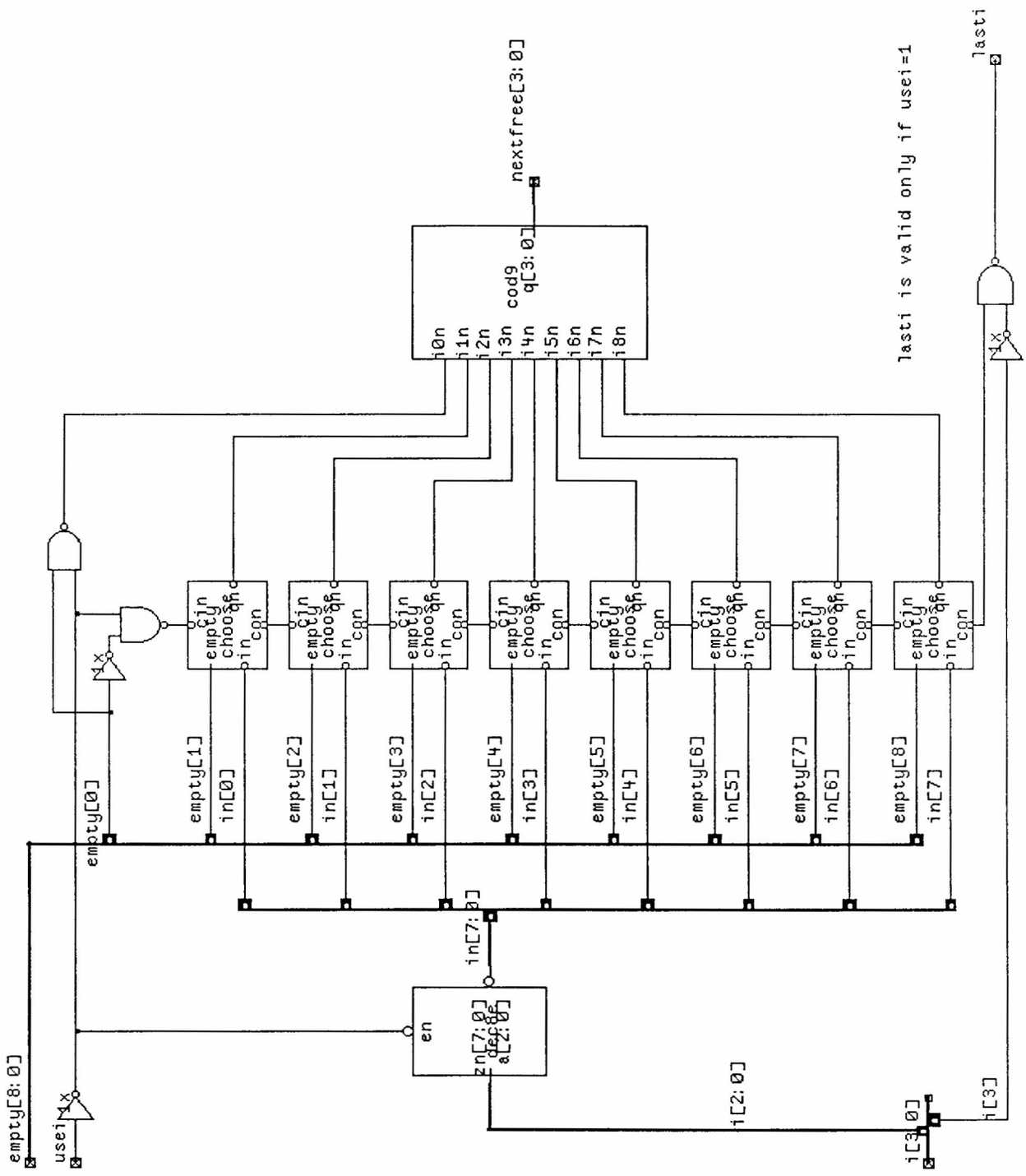
empty


e[8:0]
b[17:0]
e[8]
e[7]
e[6]
e[5]
e[4]
e[3]
e[2]
e[1]
e[0]
b[17]
b[16]
b[15]
b[14]
b[13]
b[12]
b[11]
b[10]
b[9]
b[8]
b[7]
b[6]
b[5]
b[4]
b[3]
b[2]
b[1]
b[0]

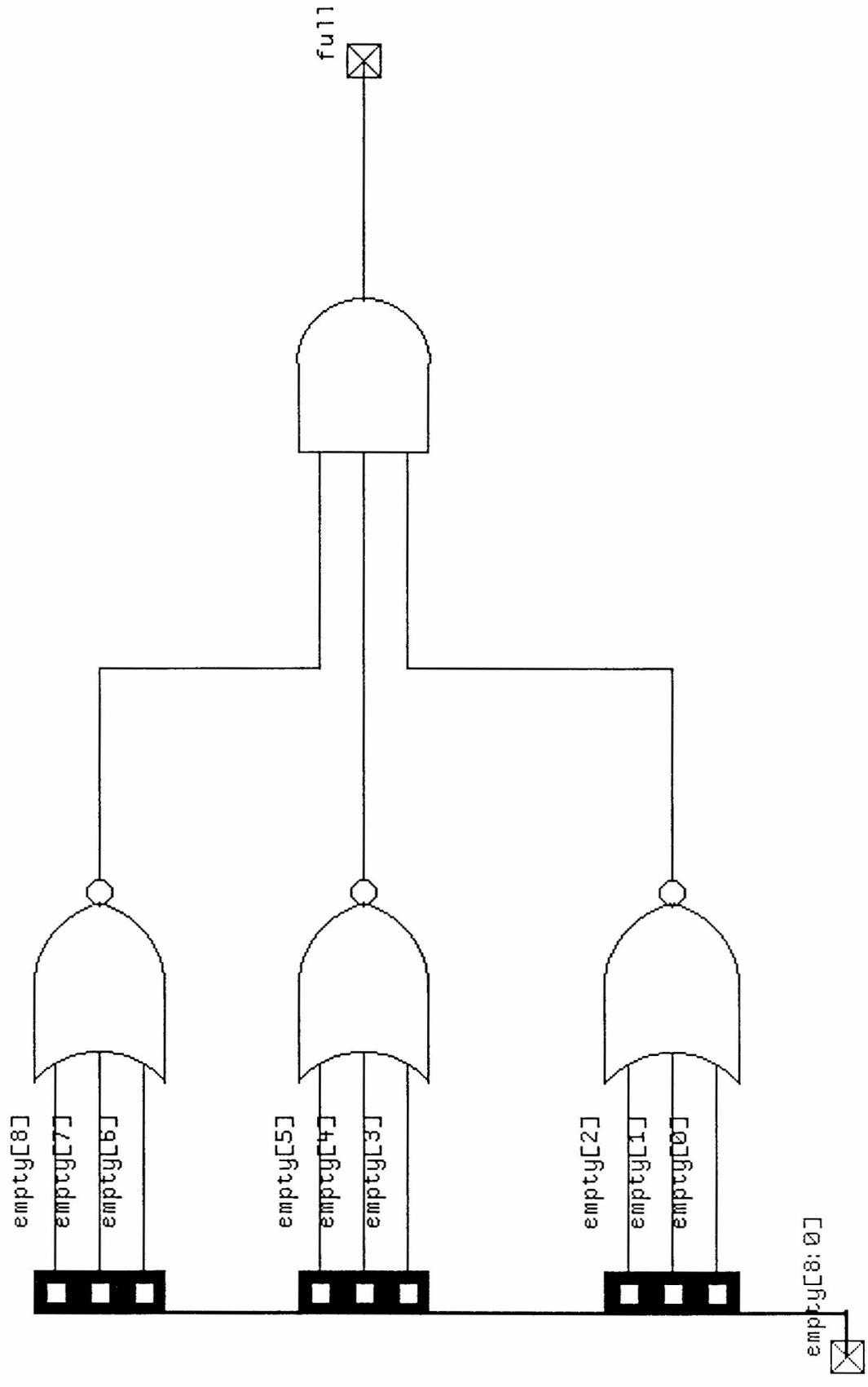
ff9


D
Q
QN
CP
D[8]
D[7]
D[6]
D[5]
D[4]
D[3]
D[2]
D[1]
D[0]
D[8:0]
Q[8]
Q[7]
Q[6]
Q[5]
Q[4]
Q[3]
Q[2]
Q[1]
Q[0]
Q[8:0]
cp

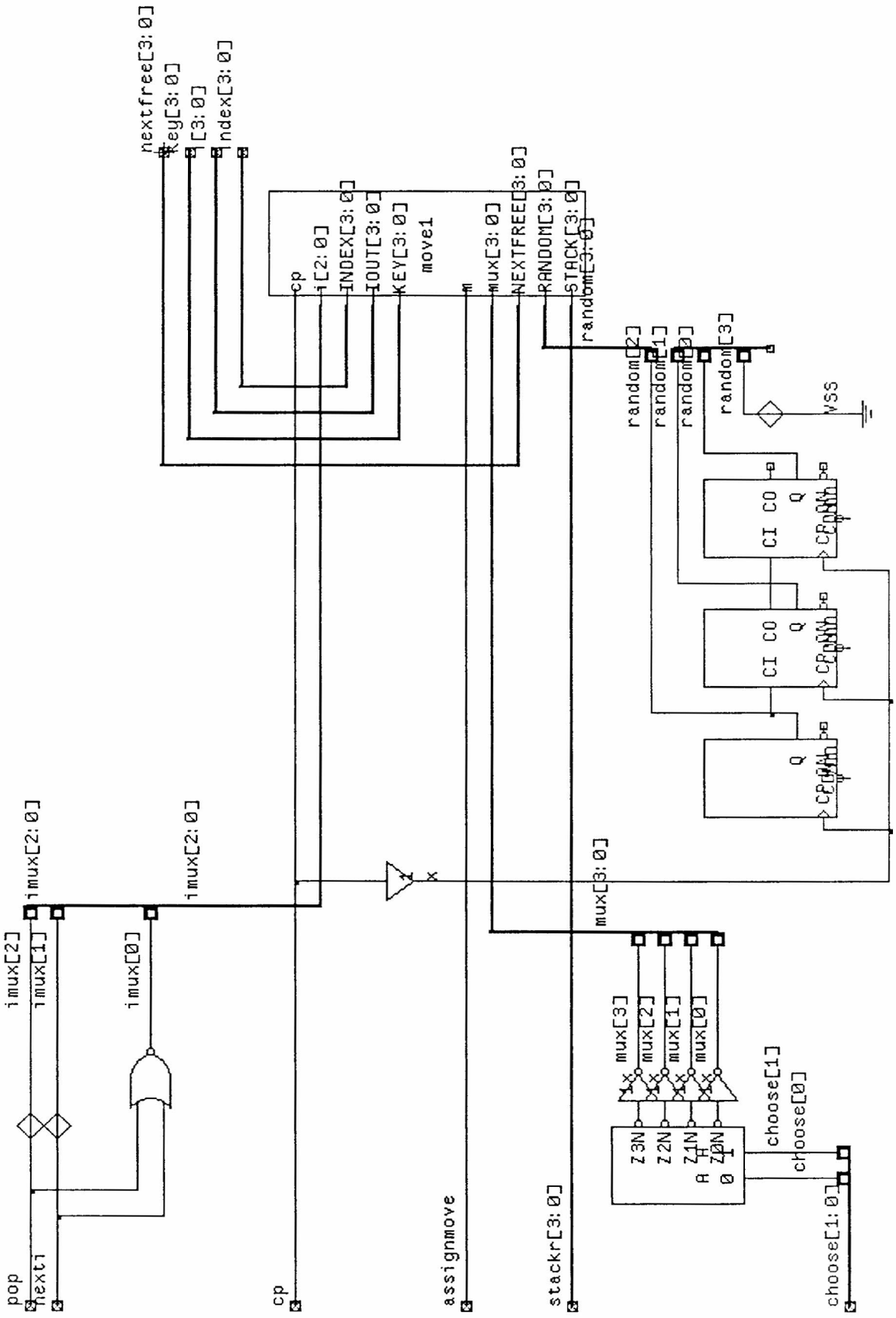
findfree


nextfree[3:0]
cod9
q[3:0]
i0n
i1n
i2n
i3n
i4n
i5n
i6n
i7n
i8n
lasti
lasti is valid only if usei=1
empty[8:0]
usei
empty[0]
empty[1]
empty[2]
empty[3]
empty[4]
empty[5]
empty[6]
empty[7]
empty[8]
in[0]
in[1]
in[2]
in[3]
in[4]
in[5]
in[6]
in[7]
in[7:0]
en
zn[7:0]
a[2:0]
i[2:0]
i[3:0]
i[3]

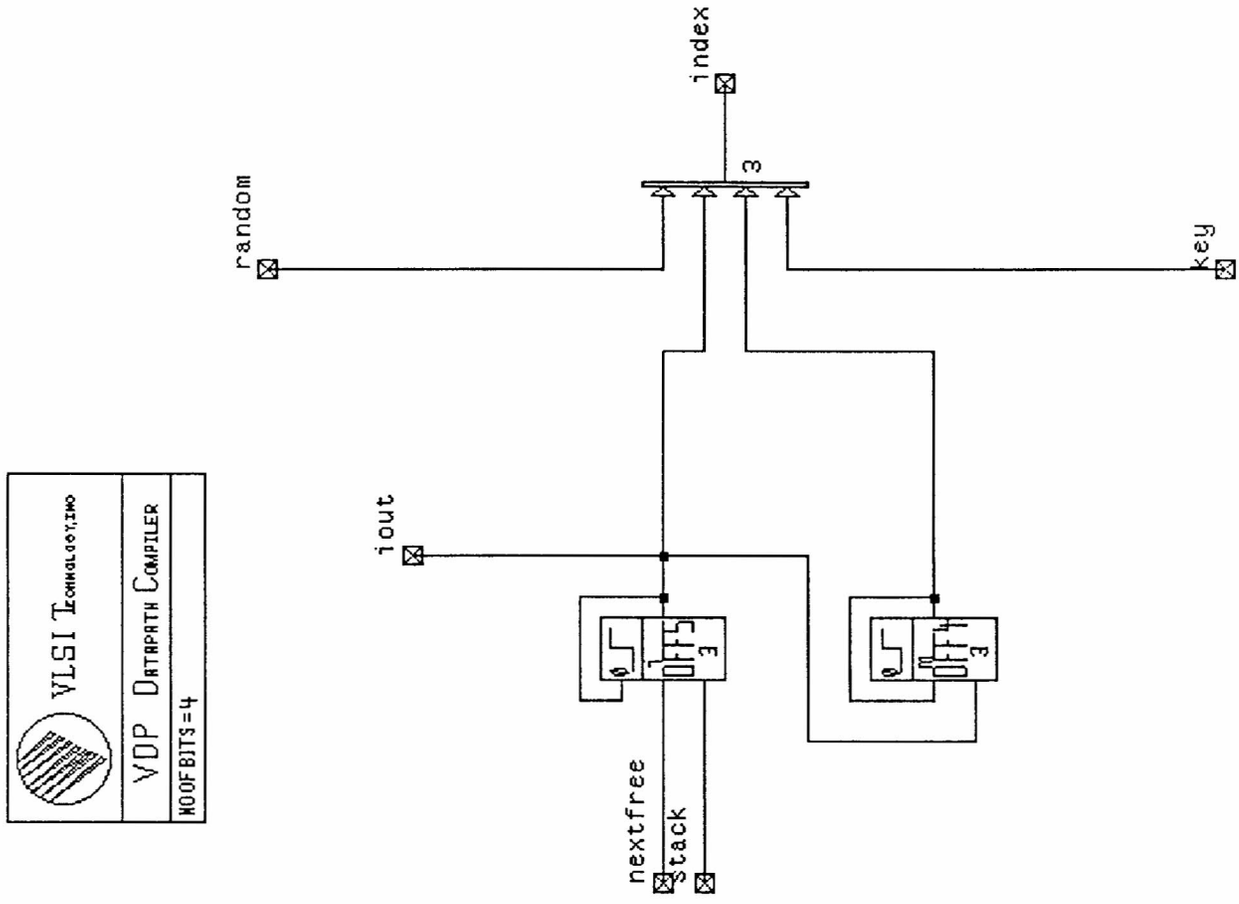
full


full
empty[8]
empty[7]
empty[6]
empty[5]
empty[4]
empty[3]
empty[2]
empty[1]
empty[0]
empty[8:0]

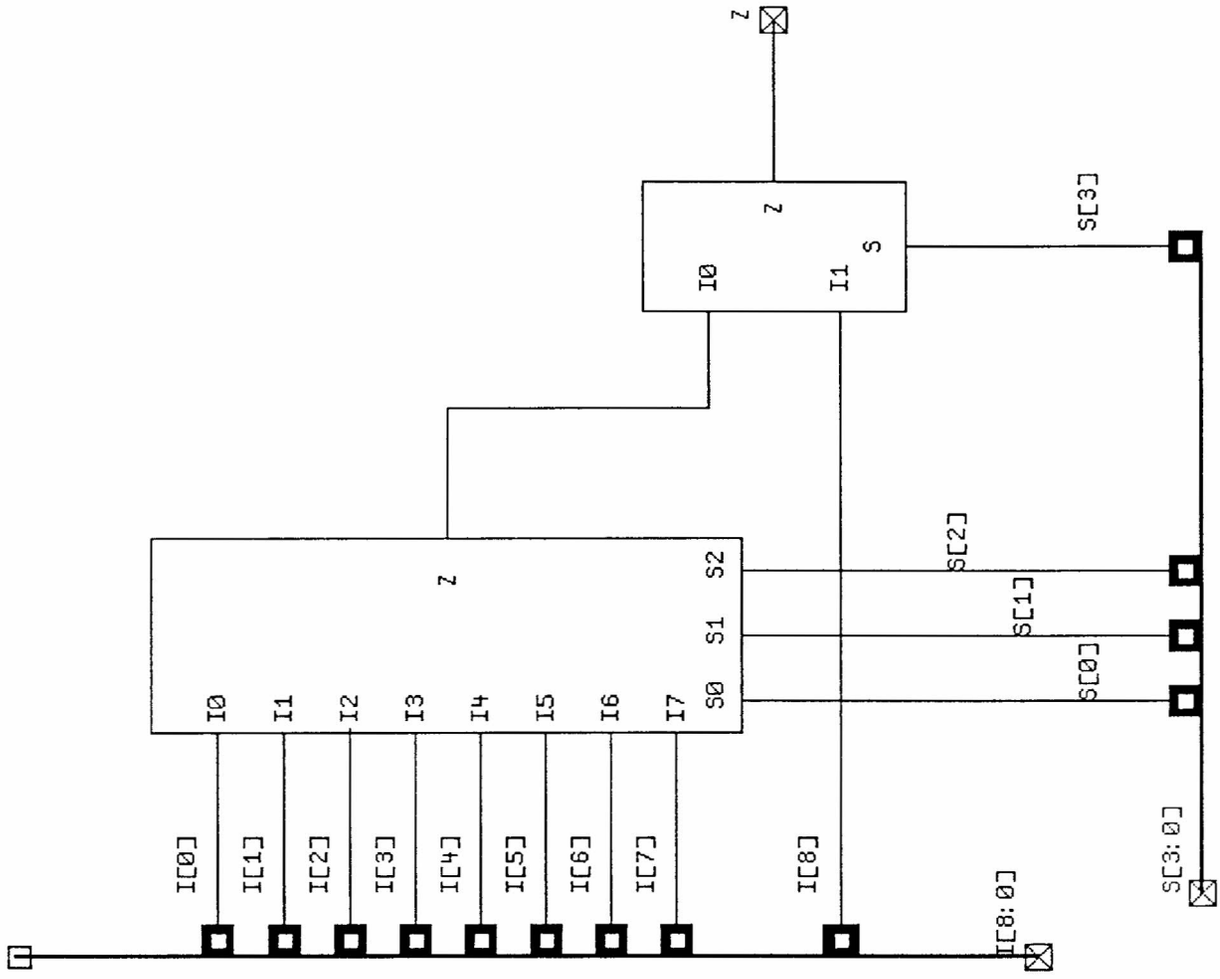
move


nextfree[3:0]
key[3:0]
[3:0]
ndex[3:0]
cp
i[2:0]
INDEX[3:0]
IOUT[3:0]
KEY[3:0]
move1
m
mux[3:0]
NEXTFREE[3:0]
RANDOM[3:0]
STACK[3:0]
random[3:0]
random[2]
random[1]
random[0]
random[3]
VSS
CI CO
Q
imux[2:0]
imux[2:0]
imux[2]
imux[1]
imux[0]
x
mux[3:0]
mux[3]
mux[2]
mux[1]
mux[0]
Z3N
Z2N
Z1N
Z0N
A
0
choose[1]
choose[0]
choose[1:0]
pop
nexti
cp
assignmove
stackr[3:0]

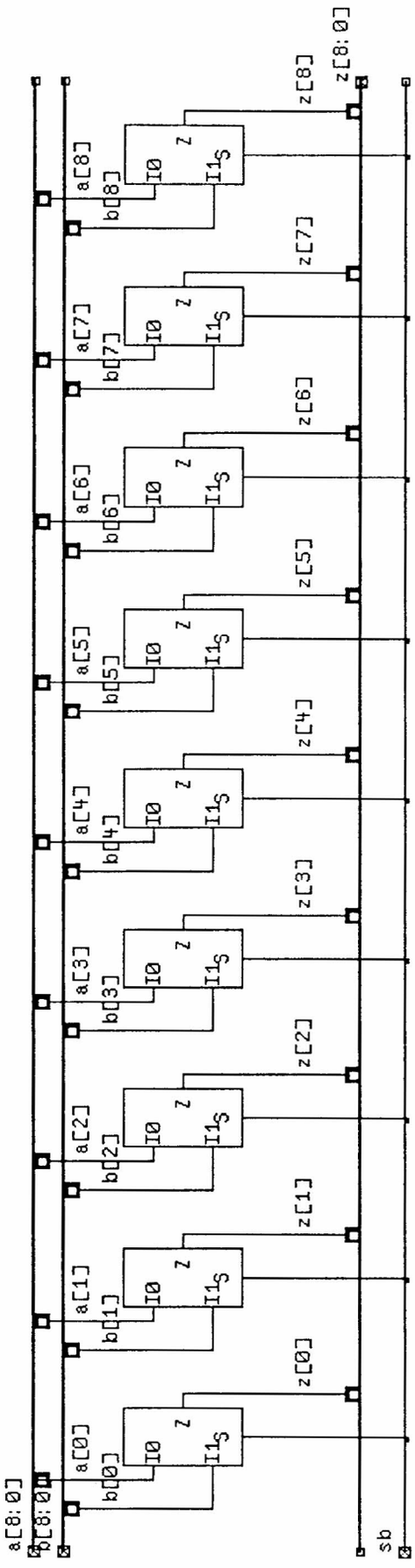
move1


index
3
random
key
iout
nextfree
stack
VLSI Technology,Inc
VDP Datapath Compiler
#OF BITS=4

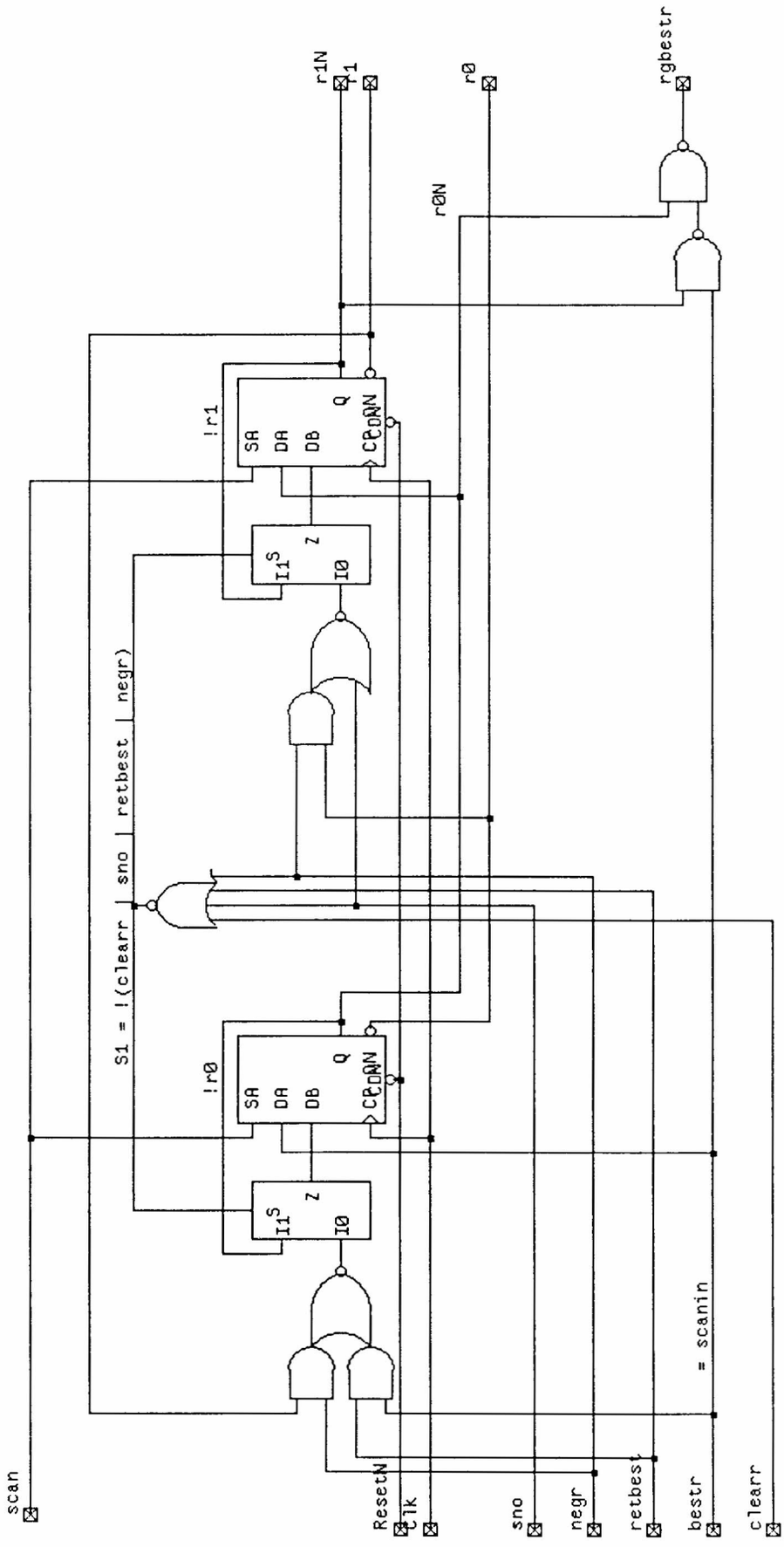
mux9


Z
Z
I0
I1
S
S[3]
Z
S2
S1
S0
S[2]
S[1]
S[0]
I0
I1
I2
I3
I4
I5
I6
I7
I[0]
I[1]
I[2]
I[3]
I[4]
I[5]
I[6]
I[7]
I[8]
I[8:0]
S[3:0]

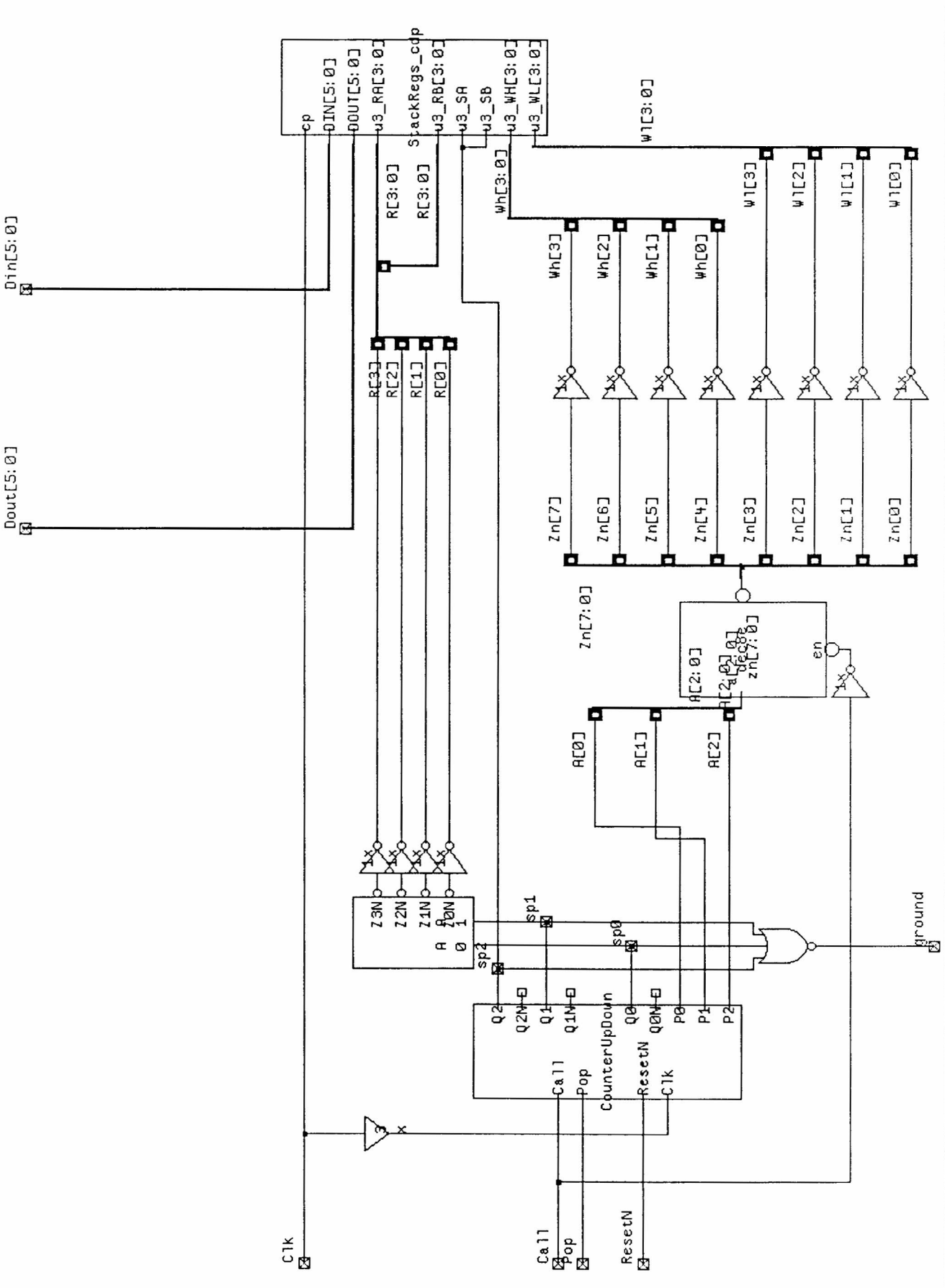
mux92

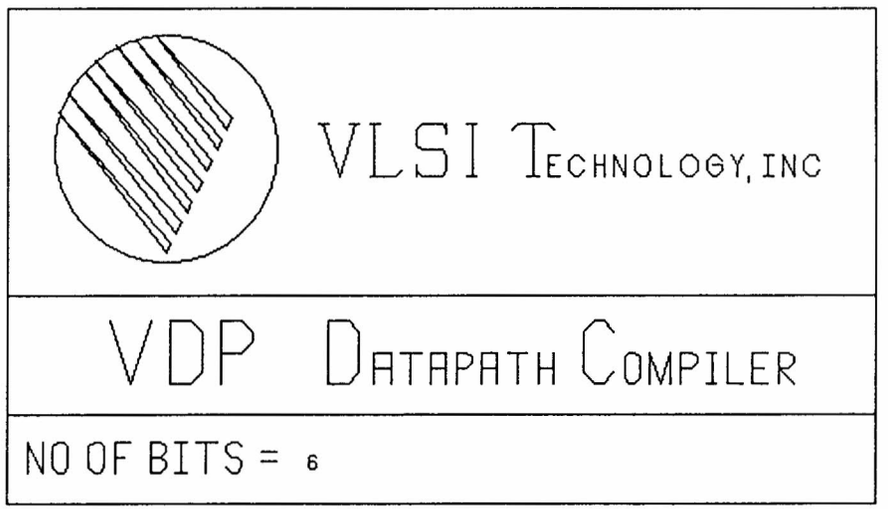

a[8:0]
b[8:0]
z[8:0]
sb
a[0]
b[0]
a[1]
b[1]
a[2]
b[2]
a[3]
b[3]
a[4]
b[4]
a[5]
b[5]
a[6]
b[6]
a[7]
b[7]
a[8]
b[8]
z[0]
z[1]
z[2]
z[3]
z[4]
z[5]
z[6]
z[7]
z[8]
I0
I1
S
Z

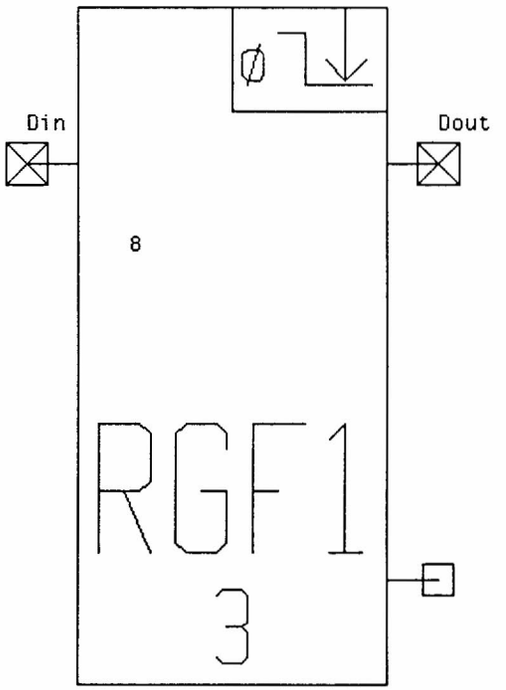
r


r1N
1
r0
rgbestr
r0N
!r1
SA
DA
DB
Q
CP
CDN
I1
S
Z
I0
S1 = !(clearr | sno | retbest | negr)
!r0
scan
ResetN
clk
sno
negr
retbest
bestr
= scanin
clearr

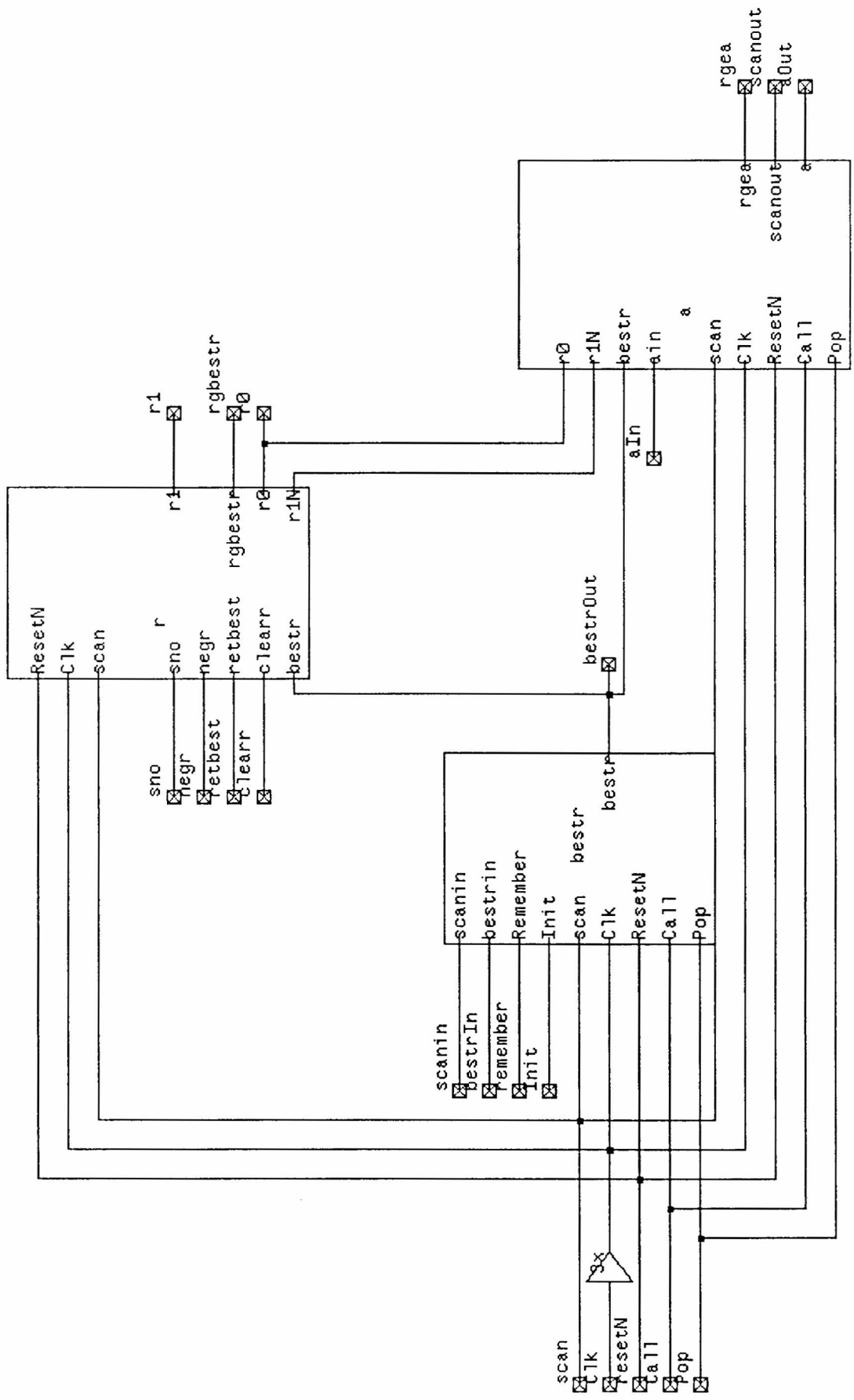
stack


StackRegs_cdp
cp
DIN[5:0]
DOUT[5:0]
u3_RA[3:0]
u3_RB[3:0]
u3_SA
u3_SB
u3_WH[3:0]
u3_WL[3:0]
Din[5:0]
Dout[5:0]
RA[3:0]
RB[3:0]
WH[3:0]
WL[3:0]
Clk
Call
Pop
ResetN
CounterUpDown
Ground

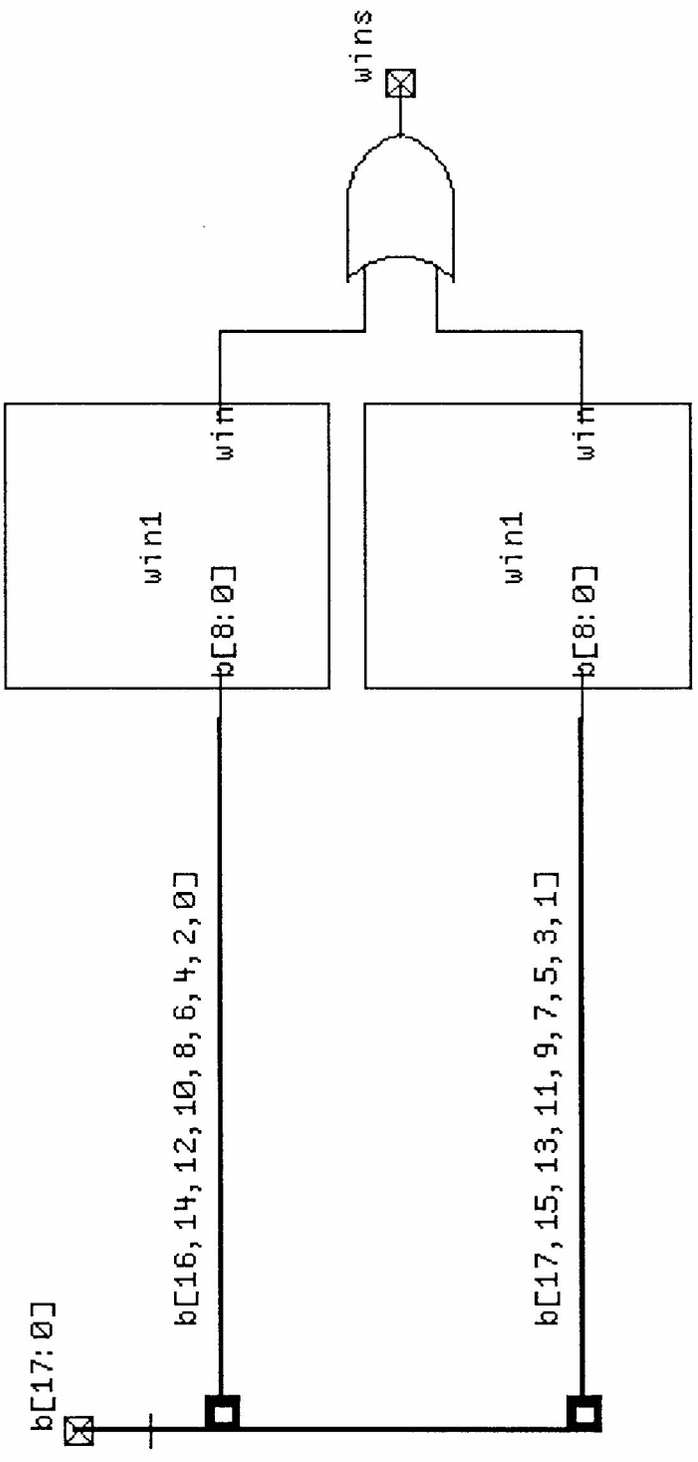

VLSI TECHNOLOGY, INC
VDP DATAPATH COMPILER
NO OF BITS = 6


Ø
Din
Dout
8
RGF1
3

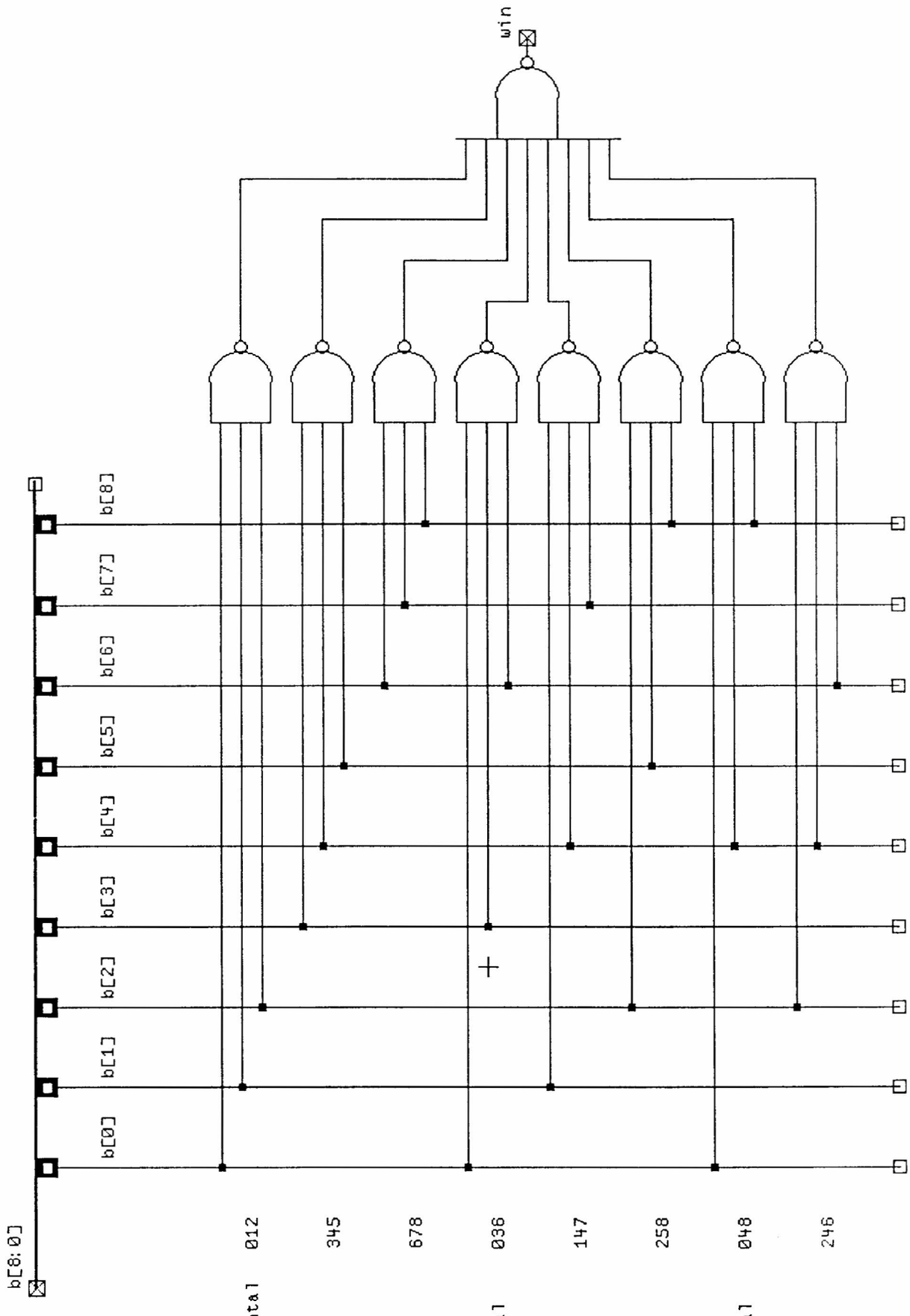
stackregs


value
rgea
scanout
aOut
a
r0
r1N
bestr
aIn
scan
Clk
ResetN
Call
Pop
r1
rgbestr
r0
aIn
a
ResetN
Clk
scan
r
sno
negr
retbest
clearr
bestr
r1
rgbestr
r0
r1N
bestrOut
sno
negr
retbest
clearr
scanin
bestrIn
Remember
Init
scan
bestr
Clk
ResetN
Call
Pop
scanin
bestrIn
Remember
Init
scan
Clk
ResetN
Call
Pop

win

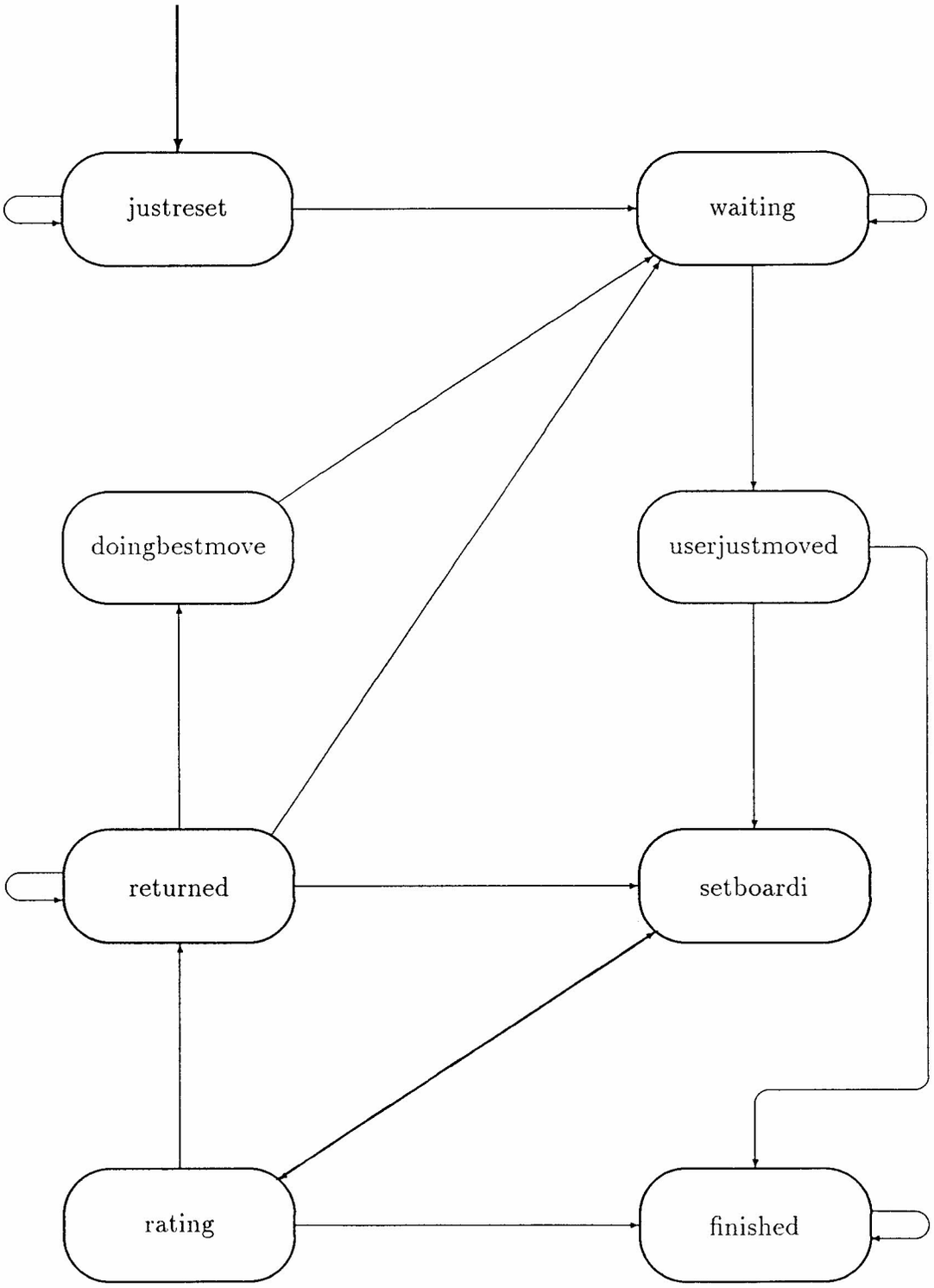

wins
win1
win
b[8:0]
win1
win
b[8:0]
b[16,14,12,10,8,6,4,2,0]
b[17,15,13,11,9,7,5,3,1]
b[17:0]

win1


win
b[8]
b[7]
b[6]
b[5]
b[4]
b[3]
b[2]
b[1]
b[0]
b[8:0]
012
345
678
036
147
258
048
246
horizontal
vertical
diagonal

```
#cell1 fsm any stateMachine 0 v7r5.4
# 19-Jan-90 16:01 19-Jan-90 16:01 vlsiii4 * .
# Finite State Machine for Move-Generator of Tic-Tac-Toe
# PMG & AM
sm fsm;

# Clock on raising edge, up to 20 MHz
clock clk 20;

reset !resetN                    --> JustReSet;

define choosetype random=00 i=01 m=10 key=11;

inputs  full ground lasti rgbestr rgea userbegins keypressed validkey wins scan scanin scanout;

outputs  assignmove=0 call=0 choose:choosetype toggleN=1 clearr=0
         fin=0 win=0 init=0 negr=0 nexti=0 pop=0 remember=0 retbest=0
         sno=0 copyboard=0 usei=0 prompt=0;

let userhasmoved = keypressed & validkey;

state JustReSet  userhasmoved                          --> justreset          choose=key !toggleN copyboard=? prompt=?,
                 userbegins                            --> waiting            copyboard=? prompt=?,
                                                       --> waiting            choose=random !toggleN copyboard=? prompt=?;

state waiting    userhasmoved                          --> userjustmoved      choose=key !toggleN copyboard=? prompt=?,
                                                       --> waiting            copyboard choose=? prompt;

state userjustmoved                             !full  --> setboardi          copyboard init nexti choose=?,
                                                 full  --> finished           copyboard=? init=? choose=?;

state setboardi                                        --> rating             choose=i !toggleN;

state rating     wins &  ground                        --> finished           copyboard=? choose=? sno=? clearr=?,
                 wins & !ground                        --> returned           choose=i !toggleN sno pop,
                 full &  ground                        --> finished           copyboard=? choose=? sno=? clearr=?,
                 full & !ground                        --> returned           choose=i !toggleN clearr pop,
                                                       --> setboardi          call nexti choose=? sno=? clearr=?;

state returned    rgea &  ground                       --> waiting            usei choose=? copyboard=? negr=?,
                  rgea & !ground                       --> returned           usei choose=i !toggleN negr pop,
                 !rgea &  rgbestr &  lasti &  ground   --> waiting            usei choose=? copyboard=? negr=?,
                 !rgea &  rgbestr &  lasti & !ground   --> returned           usei choose=i !toggleN negr=? pop,
                 !rgea &  rgbestr & !lasti & !ground   --> setboardi          usei choose=i !toggleN remember nexti negr=?,
                 !rgea &  rgbestr & !lasti &  ground   --> setboardi          usei choose=i !toggleN remember assignmove nexti negr=?,
                 !rgea & !rgbestr &  lasti &  ground   --> doingbestmove      usei choose=i !toggleN retbest=? negr=?,
                 !rgea & !rgbestr &  lasti & !ground   --> returned           usei choose=i !toggleN retbest pop,
                 !rgea & !rgbestr & !lasti             --> setboardi          usei choose=i !toggleN nexti retbest=? negr=?;

state doingbestmove                                    --> waiting            choose=m !toggleN copyboard=?;

state finished                        wins             --> finished           copyboard fin win choose=? init=?,
                                                       --> finished           copyboard fin choose=? init=?;

end
```

# C Finite State Machine



## C.1 Controller Inputs

| | |
|---|---|
| lasti | there is no empty place in board[i+1..8] |
| wins | player p wins |
| full | the board is full |
| ground | sp == 0 |
| rgea | r >= a |
| rgbestr | r > bestr |
| validmove | board[key]==empty |
| keypressed | user has pressed a key |
| userbegins | user wants to begin the game |

## C.2 Controller Outputs

| | |
|---|---|
| init | bestr=-1; |
| retbest | r=-bestr; |
| remember | bestr=0; |
| negr | r=-r; |
| sno | r=-1; |
| clearr | r=0; |
| !toggleN | if (board[choose]==empty) board[choose]=p else board[choose]=empty |
| assignmove | m=i; |
| call | push(i,bestr,a); p=otherplace(p); a=-bestr; bestr=-1; |
| pop | pop(i,bestr,a); p=otherplace(p); |
| copyboard | led=board; |
| choose[0..1] | = (random,i,m,key) |
| nexti | i=nextfree(li), li= (-1,i) |
| usei | li=i |
| fin | the game is finished |
| win | the game is finished and the chip has won |
| prompt | the chip is waiting for the user's move |

## C.3 Reset

| | |
|---|---|
| reset | p=computer; a=1; board[0..17]=empty |

# D Pin assignment

| | | | | | | |
|---|---|---|---|---|---|---|
| not connected | | 1 | | 64 | P | Vdd core |
| keyboardN [illegible] | I | 2 | | 63 | | not connected |
| userbegins | I | 3 | | 62 | O | copyboard |
| scan | I | 4 | | 61 | O | sp 2 |
| win | I | 5 | | 60 | O | led [illegible] |
| sp 0 | I | 6 | | 59 | O | sp 1 |
| piN | I | 7 | | 58 | O | fin |
| not connected | | 8 | | 57 | | not connected |
| scanin | I | 9 | | 56 | O | scanout |
| resetN | I | 10 | | 55 | O | prompt |
| not connected | | 11 | | 54 | | not connected |
| Vdd pads | P | 12 | | 53 | | not connected |
| not connected | | 13 | | 52 | | not connected |
| Vss pads | P | 14 | | 51 | O | led [illegible] |
| not connected | | 15 | | 50 | O | led 11 |
| keyboardN [illegible] | I | 16 | | 49 | O | led 7 |
| keyboardN [illegible] | I | 17 | | 48 | O | led 9 |
| keyboardN [illegible] | I | 18 | | 47 | O | led 0 |
| not connected | | 19 | | 46 | O | led 6 |
| cp | I | 20 | | 45 | O | led [illegible] |
| not connected | | 21 | | 44 | O | led [illegible] |
| i 0 | O | 22 | | 43 | O | led [illegible] |
| keyboardN [illegible] | I | 23 | | 42 | O | led 16 |
| keyboardN [illegible] | I | 24 | | 41 | O | led 15 |
| i 1 | O | 25 | | 40 | O | led 1 |
| i 2 | O | 26 | | 39 | O | led 12 |
| i 3 | O | 27 | | 38 | O | led [illegible] |
| keyboardN [illegible] | I | 28 | | 37 | O | led 17 |
| keyboardN [illegible] | I | 29 | | 36 | O | led 14 |
| keyboardN [illegible] | I | 30 | | 35 | O | led 13 |
| not connected | | 31 | | 34 | | not connected |
| Vss core | P | 32 | | 33 | | not connected |

I – Input signal
O – Output signal
P – Power supply

# E Simulation

This appendix contains the input and the output of a sample VTIQsim simulation. In the first few cycles, a Tic-Tac-Toe position is input. 1300 nanoseconds after the start of the simulation, the user begins with his first move.

VTIQsim is not able to output symbolic state names from a finite state machine. Thus we have written a C program computing the state name from the state register value and appending it to the end of every line.

## E.1 Simulation Input

```
#cell1 report any sim 0 v7r5.4
# 9-Jan-90 12:45 9-Jan-90 12:45 vlsiii4 * .
mode qsim
trace (static,tabular)

switch sim

load [fns]chipsim

#input: resetN cp userbegins keyboardN[8:0]
#test input: scan scanin pin
#output: led[17:0] fin win prompt copyboard
#test output: i[3:0] sp2 sp1 sp0

vector keyboardN[8:0]
vector led[17:0]
vector sp[2:0] sp2 sp1 sp0
vector core.board[17:0]
vector core.fsm.choose[1:0]
vector i[3:0]
vector statereg core.fsm.sr1.q core.fsm.sr2.q core.fsm.sr3.q

radix 16
radix 2 led keyboardN

watch resetN cp keyboardN statereg
watch core.fsm.wins core.fsm.choose core.fsm.toggleN
watch core.fsm.usei core.fsm.nexti
watch sp i fin win prompt led copyboard

clock cp 0(100) 1(100)

l resetN
l scan
l scanin
l pin
h userbegins
```

```
inputs 'b111111111 keyboardN

output report
trace (static,tabular)

# position:  OX                X: user's stone
#           XOO                O: chip's stone
#            X

#user
l pin
inputs 'b111111011 keyboardN
cycle
h resetN
inputs 'b111110111 keyboardN
cycle

#chip
inputs 'b111111101 keyboardN
cycle
h pin
inputs 'b111101111 keyboardN
cycle
inputs 'b111011111 keyboardN
cycle
inputs 'b111111111 keyboardN
cycle
l pin
l keyboardN[7]
cycle
h keyboardN[7]

cycle 36

l keyboardN[8]
cycle
h keyboardN[8]

cycle 6
```

## E.2 Simulation Output

```
#
#
#          R C k          s c c c c c c s i F W P l                  C
#          E P e          t o o o o o o p   I I R e                  O
#          S   y          a r r r r r r     N N O d                  P
#          E   b          t e e e e e e                M            Y
#          T   o          e . . . . . .                P            B
#          N   a          r f f f f f f                T            O
#              r          e s s s s s s                             A
#              d          g m m m m m m                             R
#              N            . . . . . .                             D
#                         W c T U N
#              [      [   I h O S E           [              [
#              8      0   N o G E X           1              0
#              ]      ]   S o G I T           7              [
#                           s L   I           ]
#                         e E
#                           N
#
     100.0   0 0 111111011 3 0 u u 0 0 0 u 0 0 0 000000000000000000 0
     200.0   0 1 111111011 3 0 3 0 0 0 0 u 0 0 0 000000000000000000 0 justreset
     300.0   1 0 111110111 3 0 3 0 0 0 0 u 0 0 0 000000000000000000 0
     400.0   1 1 111110111 3 0 3 0 0 0 0 u 0 0 0 000000000000100000 0 justreset
     500.0   1 0 111111101 3 0 3 0 0 0 0 u 0 0 0 000000000000100000 0
     600.0   1 1 111111101 3 0 3 0 0 0 0 u 0 0 0 000000000010100000 0 justreset
     700.0   1 0 111101111 3 0 3 0 0 0 0 u 0 0 0 000000000010100000 0
     800.0   1 1 111101111 3 0 3 0 0 0 0 u 0 0 0 000000000010100100 0 justreset
     900.0   1 0 111011111 3 0 3 0 0 0 0 u 0 0 0 000000000010100100 0
    1000.0   1 1 111011111 3 0 3 0 0 0 0 u 0 0 0 000000000110100100 0 justreset
    1100.0   1 0 111111111 3 0 3 0 0 0 0 u 0 0 0 000000000110100100 0
    1200.0   1 1 111111111 3 0 0 1 0 0 0 u 0 0 0 000000010110100100 0 justreset
    1300.0   1 0 101111111 3 0 0 1 0 0 0 u 0 0 0 000000010110100100 0
    1400.0   1 1 101111111 0 0 3 0 0 0 0 u 0 0 1 000000010110100100 0 waiting
    1500.0   1 0 111111111 0 0 3 0 0 0 0 u 0 0 1 000000010110100100 0
    1600.0   1 1 111111111 5 0 0 1 0 1 0 u 0 0 0 001000010110100100 1 userjustmoved
    1700.0   1 0 111111111 5 0 0 1 0 1 0 u 0 0 0 001000010110100100 1
    1800.0   1 1 111111111 7 0 1 0 0 0 0 0 0 0 0 001000010110100100 0 setboardi
    1900.0   1 0 111111111 7 0 1 0 0 0 0 0 0 0 0 001000010110100100 0
    2000.0   1 1 111111111 2 0 0 1 0 1 0 0 0 0 0 001000010110100101 0 rating
    2100.0   1 0 111111111 2 0 0 1 0 1 0 0 0 0 0 001000010110100101 0
    2200.0   1 1 111111111 7 0 1 0 0 0 1 6 0 0 0 001000010110100101 0 setboardi
    2300.0   1 0 111111111 7 0 1 0 0 0 1 6 0 0 0 001000010110100101 0
    2400.0   1 1 111111111 2 0 0 1 0 1 1 6 0 0 0 001010010110100101 0 rating
    2500.0   1 0 111111111 2 0 0 1 0 1 1 6 0 0 0 001010010110100101 0
    2600.0   1 1 111111111 7 0 1 0 0 0 2 8 0 0 0 001010010110100101 0 setboardi
    2700.0   1 0 111111111 7 0 1 0 0 0 2 8 0 0 0 001010010110100101 0
    2800.0   1 1 111111111 2 1 1 0 0 0 2 8 0 0 0 011010010110100101 0 rating
    2900.0   1 0 111111111 2 1 1 0 0 0 2 8 0 0 0 011010010110100101 0
    3000.0   1 1 111111111 1 0 1 0 1 1 1 6 0 0 0 001010010110100101 0 returned
    3100.0   1 0 111111111 1 0 1 0 1 1 1 6 0 0 0 001010010110100101 0
    3200.0   1 1 111111111 7 0 1 0 0 0 1 8 0 0 0 001000010110100101 0 setboardi
```

```
3300.0   1 0 111111111 7 0 1 0 0 0 1 8 0 0 0 001000010110100101 0
3400.0   1 1 111111111 2 0 0 1 0 1 1 8 0 0 0 101000010110100101 0 rating
3500.0   1 0 111111111 2 0 0 1 0 1 1 8 0 0 0 101000010110100101 0
3600.0   1 1 111111111 7 0 1 0 0 0 2 6 0 0 0 101000010110100101 0 setboardi
3700.0   1 0 111111111 7 0 1 0 0 0 2 6 0 0 0 101000010110100101 0
3800.0   1 1 111111111 2 0 1 0 0 0 2 6 0 0 0 101001010110100101 0 rating
3900.0   1 0 111111111 2 0 1 0 0 0 2 6 0 0 0 101001010110100101 0
4000.0   1 1 111111111 1 0 1 0 1 0 1 8 0 0 0 101000010110100101 0 returned
4100.0   1 0 111111111 1 0 1 0 1 0 1 8 0 0 0 101000010110100101 0
4200.0   1 1 111111111 1 0 1 0 1 1 0 0 0 0 0 001000010110100101 0 returned
4300.0   1 0 111111111 1 0 1 0 1 1 0 0 0 0 0 001000010110100101 0
4400.0   1 1 111111111 7 0 1 0 0 0 0 6 0 0 0 001000010110100100 0 setboardi
4500.0   1 0 111111111 7 0 1 0 0 0 0 6 0 0 0 001000010110100100 0
4600.0   1 1 111111111 2 0 0 1 0 1 0 6 0 0 0 001001010110100100 0 rating
4700.0   1 0 111111111 2 0 0 1 0 1 0 6 0 0 0 001001010110100100 0
4800.0   1 1 111111111 7 0 1 0 0 0 1 0 0 0 0 001001010110100100 0 setboardi
4900.0   1 0 111111111 7 0 1 0 0 0 1 0 0 0 0 001001010110100100 0
5000.0   1 1 111111111 2 0 0 1 0 1 1 0 0 0 0 001001010110100110 0 rating
5100.0   1 0 111111111 2 0 0 1 0 1 1 0 0 0 0 001001010110100110 0
5200.0   1 1 111111111 7 0 1 0 0 0 2 8 0 0 0 001001010110100110 0 setboardi
5300.0   1 0 111111111 7 0 1 0 0 0 2 8 0 0 0 001001010110100110 0
5400.0   1 1 111111111 2 0 1 0 0 0 2 8 0 0 0 011001010110100110 0 rating
5500.0   1 0 111111111 2 0 1 0 0 0 2 8 0 0 0 011001010110100110 0
5600.0   1 1 111111111 1 0 1 0 1 0 1 0 0 0 0 001001010110100110 0 returned
5700.0   1 0 111111111 1 0 1 0 1 0 1 0 0 0 0 001001010110100110 0
5800.0   1 1 111111111 1 0 1 0 1 1 0 6 0 0 0 001001010110100100 0 returned
5900.0   1 0 111111111 1 0 1 0 1 1 0 6 0 0 0 001001010110100100 0
6000.0   1 1 111111111 7 0 1 0 0 0 0 8 0 0 0 001000010110100100 0 setboardi
6100.0   1 0 111111111 7 0 1 0 0 0 0 8 0 0 0 001000010110100100 0
6200.0   1 1 111111111 2 0 0 1 0 1 0 8 0 0 0 011000010110100100 0 rating
6300.0   1 0 111111111 2 0 0 1 0 1 0 8 0 0 0 011000010110100100 0
6400.0   1 1 111111111 7 0 1 0 0 0 1 0 0 0 0 011000010110100100 0 setboardi
6500.0   1 0 111111111 7 0 1 0 0 0 1 0 0 0 0 011000010110100100 0
6600.0   1 1 111111111 2 0 0 1 0 1 1 0 0 0 0 011000010110100110 0 rating
6700.0   1 0 111111111 2 0 0 1 0 1 1 0 0 0 0 011000010110100110 0
6800.0   1 1 111111111 7 0 1 0 0 0 2 6 0 0 0 011000010110100110 0 setboardi
6900.0   1 0 111111111 7 0 1 0 0 0 2 6 0 0 0 011000010110100110 0
7000.0   1 1 111111111 2 0 1 0 0 0 2 6 0 0 0 011001010110100110 0 rating
7100.0   1 0 111111111 2 0 1 0 0 0 2 6 0 0 0 011001010110100110 0
7200.0   1 1 111111111 1 0 1 0 1 0 1 0 0 0 0 011000010110100110 0 returned
7300.0   1 0 111111111 1 0 1 0 1 0 1 0 0 0 0 011000010110100110 0
7400.0   1 1 111111111 1 0 1 0 1 0 0 8 0 0 0 011000010110100100 0 returned
7500.0   1 0 111111111 1 0 1 0 1 0 0 8 0 0 0 011000010110100100 0
7600.0   1 1 111111111 6 0 2 0 0 0 0 8 0 0 0 001000010110100100 0 doingbestmove
7700.0   1 0 111111111 6 0 2 0 0 0 0 8 0 0 0 001000010110100100 0
7800.0   1 1 111111111 0 0 0 1 0 0 0 8 0 0 1 001000010110100101 1 waiting
7900.0   1 0 111111111 0 0 0 1 0 0 0 8 0 0 1 001000010110100101 1
8000.0   1 1 111111111 0 0 0 1 0 0 0 8 0 0 1 001000010110100101 1 waiting
8100.0   1 0 111111111 0 0 0 1 0 0 0 8 0 0 1 001000010110100101 1
8200.0   1 1 111111111 0 0 0 1 0 0 0 8 0 0 1 001000010110100101 1 waiting
8300.0   1 0 111111111 0 0 0 1 0 0 0 8 0 0 1 001000010110100101 1
8400.0   1 1 111111111 0 0 0 1 0 0 0 8 0 0 1 001000010110100101 1 waiting
8500.0   1 0 111111111 0 0 0 1 0 0 0 8 0 0 1 001000010110100101 1
```

```
 8600.0   1 1 111111111 0 0 0 1 0 0 0 8 0 0 1 001000010110100101 1 waiting
 8700.0   1 0 011111111 0 0 0 1 0 0 0 8 0 0 1 001000010110100101 1
 8800.0   1 1 011111111 0 0 3 0 0 0 0 8 0 0 1 001000010110100101 0 waiting
 8900.0   1 0 111111111 0 0 3 0 0 0 0 8 0 0 1 001000010110100101 0
 9000.0   1 1 111111111 5 0 0 1 0 1 0 8 0 0 0 101000010110100101 1 userjustmoved
 9100.0   1 0 111111111 5 0 0 1 0 1 0 8 0 0 0 101000010110100101 1
 9200.0   1 1 111111111 7 0 1 0 0 0 0 6 0 0 0 101000010110100101 0 setboardi
 9300.0   1 0 111111111 7 0 1 0 0 0 0 6 0 0 0 101000010110100101 0
 9400.0   1 1 111111111 2 0 0 1 0 0 0 6 0 0 0 101001010110100101 0 rating
 9500.0   1 0 111111111 2 0 0 1 0 0 0 6 0 0 0 101001010110100101 0
 9600.0   1 1 111111111 4 0 0 1 0 0 0 6 1 0 0 101001010110100101 1 finished
 9700.0   1 0 111111111 4 0 0 1 0 0 0 6 1 0 0 101001010110100101 1
 9800.0   1 1 111111111 4 0 0 1 0 0 0 6 1 0 0 101001010110100101 1 finished
 9900.0   1 0 111111111 4 0 0 1 0 0 0 6 1 0 0 101001010110100101 1
10000.0   1 1 111111111 4 0 0 1 0 0 0 6 1 0 0 101001010110100101 1 finished
```

# F C Program

This appendix lists a C program playing Tic-Tac-Toeusing the same algorithm as the chip.

## F.1 Header File

```
/* ttmain.h */

#define SIZE 9                  /* number of places on the board */

enum place {empty, computer, human};
#define otherplace(p) (3-p)
#define true 1
#define false 0

typedef int value; /* -1..1 */
typedef int boolean; /* 0..1 */
typedef char Number; /* 0..8 */
typedef enum place BoardTyp [9];
typedef char* string;

extern BoardTyp Board;
extern int sp;
```

## F.2 Algorithm

```
/* alg.c: Algorithms for Tic-Tac-Toe */

#include "ttmain.h"
#define ground (sp==0)
#define MAXSP 8

static Number m;                /* the algorithm has chosen this place */
static value r;
static struct astack { int st_p, st_a, st_bestr, st_i; } stack[MAXSP];
static int sp;                  /* the stack pointer */
static enum place p;
static value a;
static value bestr;             /* bestr in -1..0; a in 0..1 */
static Number i;

Number Move(pp)
     enum place pp;
{
  m=-1;                         /* only for the C program */
  sp=0;
  p=pp;
```

```
  a=1;
  rek2();
  if (m==-1) {
    puts("no move found\n");
    exit(1);
  }
  if (sp!=0) {
    puts("stack error");
    exit(1);
  }
  return m;
}

push()
{
  if (sp==MAXSP) {
    puts("stack overflow");
    exit(1);
  }
  stack[sp].st_p = p;
  stack[sp].st_a = a;
  stack[sp].st_bestr = bestr;
  stack[sp].st_i = i;
  sp++;
}

pop()
{
  --sp;
  p = stack[sp].st_p;
  a = stack[sp].st_a;
  bestr = stack[sp].st_bestr;
  i = stack[sp].st_i;
}

rek2()                          /* find the best move for player p */
                                /* or any move with value >= a */
                                /* if (ground) store that move in m */
                                /* precondition: the board is not full */
                                /*               there exists a move with */
                                /*               value > -1 */
{
  bestr=-1;
  for(i=0;i<SIZE;i++) {
    if (Board[i]==empty) {
      Board[i]=p;

      if (wins(p)) {            /* this move makes three in a row */
        if (ground) m=i;
        Board[i]=empty;
        r=-1; return;
      }
```

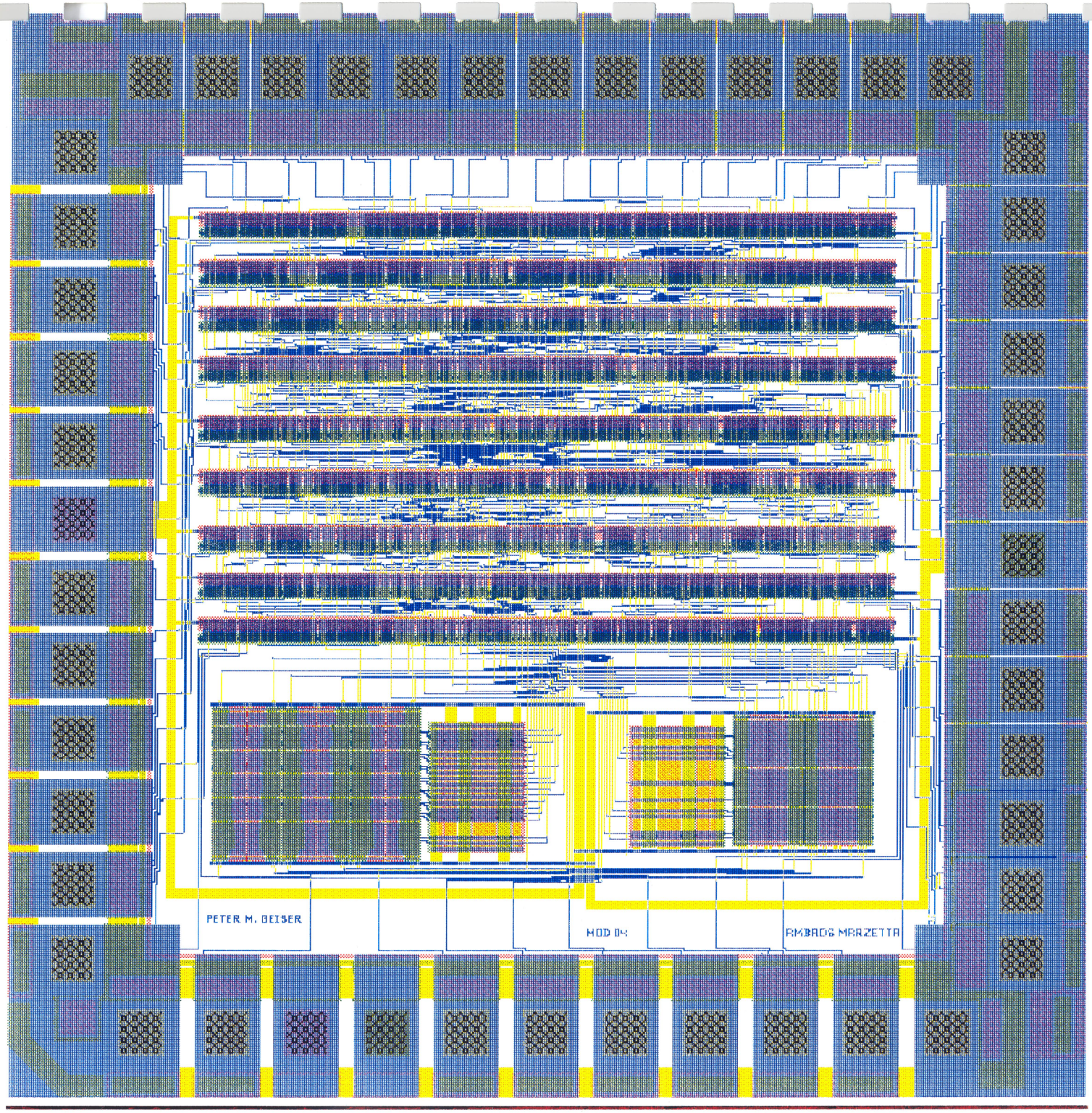
```
      if (full()) {             /* this was the last move (tie) */
        if (ground) m=i;
        Board[i]=empty;
        r=0; return;
      }

      push();
      p=otherplace(p);
      a=-bestr;
      rek2();                   /* this is the recursive call */
      pop();

      Board[i]=empty;

      if (r>=a) {               /* this move has a value >= a */
        if (ground) m=i;        /* (cut the search tree) */
        r=-r; return;
      }

      if (r>bestr) {            /* this move is better than the other moves */
        bestr=r;                /* remember it and its value */
        if (ground) m=i;
      }
    }
  }
  r=-bestr;
}

wins(p)                         /* has player p three in a row? */
     enum place p;
{
#define b(i) (Board[i]==p)

  return (b(0) && (b(1)&&b(2) || b(3)&&b(6) || b(4)&&b(8)) ||
          b(4) && (b(1)&&b(7) || b(2)&&b(6) || b(3)&&b(5)) ||
          b(8) && (b(6)&&b(7) || b(2)&&b(5)));
}

full()                          /* are all nine fields filled? */
{
  Number i;

  for(i=0;i<SIZE;i++) if (!Board[i]) return 0;
  return 1;
}
```

## F.3 Main Program

```
/* Tic-Tac-Toe main program */

#include <ctype.h>
#include <curses.h>
#include "ttmain.h"

BoardTyp Board;


int timeline;
char levch;
Number mv;
long time;
FILE *tf;

main()
{
  tf=fopen("alg/test","a+");
  do {
    reset();
    fprintf(tf,"\n");
    if(askmessage("Do you want to play first?")) {
      fprintf(tf,"%d      ",askmove());
      levch++;
    }
    for(;;) {
      levch++;
      time=clock();
      mv=Move();
      time=(clock()-time)/1000;
      showtime(time);
      placenewstone(mv, computer);
      fprintf(tf,"%d%6d ",mv,time);
      if (wins(computer)||full()) break;
      fprintf(tf,"%d      ",askmove());
      levch++;
      if (wins(human)||full()) break;
    }
    showscore();
  } while (askmessage("Do you want to play again? "));
}

char yesorno()
{
  char ch;
  refresh();
  do {
    ch=toupper(getch());
  } while (ch!='Y' && ch!='N');
  return ch=='Y';
}
```

```
placenewstone ( i, stone )
     Number i;
     enum place stone;
{
  Board[i]=stone;
  move((i/3)*2+3, (i%3)*4+4);
  if(stone==human) addch('X');
  else
    if(stone==computer) addch('O');
    else
      addch(' ');
}

showtime(t)
     long t;
{
  move(timeline++,0);
  printw("Time for %c. move :%8d milliseconds.",levch,t);
}

showboard()
{
  move(2,0);
  addstr("  !---!---!---!\n");
  addstr("1 !   !   !   !\n");
  addstr("  !---!---!---!\n");
  addstr("2 !   !   !   !\n");
  addstr("  !---!---!---!\n");
  addstr("3 !   !   !   !\n");
  addstr("  !---!---!---!\n");
  addstr("    A   B   C");
}

showscore()
{
  move(12,0);
  if(wins(human)) addstr("You won.");
  else
    if(wins(computer)) addstr("I won.");
    else addstr("It's a draw");
}

reset()
{
  Number i;

  /* used on chip */
  for(i=0;i<9;i++) Board[i]=empty;
  sp=0;

  /* used only in C program */
  initscr();
```

```
  clear();
  move(0,0);
  addstr("T I C - T A C - T O E");
  showboard();
  timeline=16;
  levch='0';
}

int askmessage ( str )
     string str;
{
  char a;
  move(14,0);
  addstr(str);
  addstr(" (Y/N)");
  a=yesorno();
  move(14,0);
  clrtoeol();
  refresh();
  return a;
}

int askmove()
{
  char c;
  Number i;
  char j=false;

  do {
    move(14,0);
    addstr("Please enter your move : ");
    refresh();
    c=toupper(getch());
    if((c=='A') || (c=='B') || (c=='C')) {
      i=c-'A';
      c=getch();
      if((c=='1') || (c=='2') || (c=='3')) {
        i+=3*(c-'1');
        if(Board[i]==empty) {
          placenewstone(i,human);
          j=true;
        }
      }
    }
  } while(!j);
  refresh();
  return i;
}
```


PETER M. GEISER
MOD 04
AMBROS MARZETTA