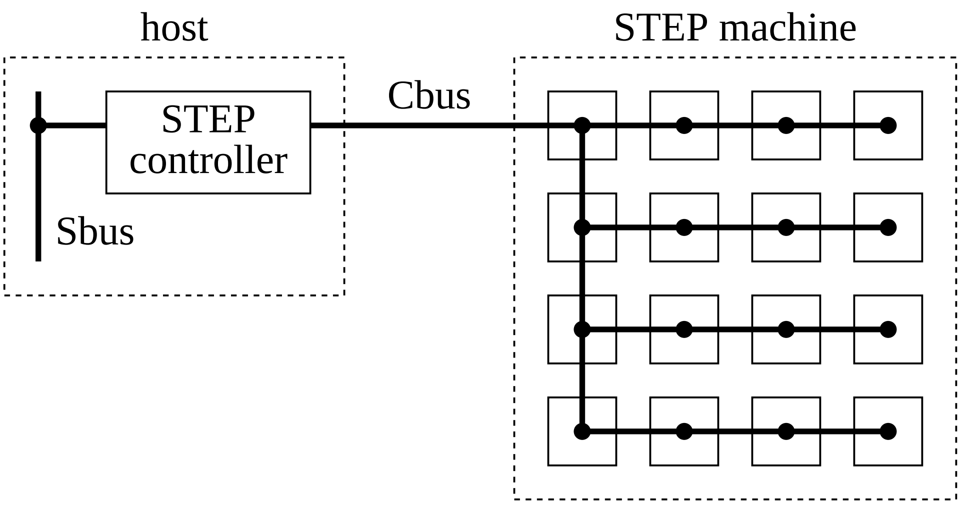
Norman Margolus
Tommaso Toffoli

STEP: a SpaceTime Event Processor

The STEP controller

(version 2.2.3)



MIT Laboratory for Computer Science
Cambridge, MA 02139

Contents

1 The controller as a steplist processor

Commands to the STEP machine are given by writing in host memory instructions for the STEP controller, and appropriately directing to controller to locate and interpret these instructions. Here we describe (a) the format of these instructions; (b) the protocol by which the controller is called to execute them and by which it acknowledges completion—or signals exceptions encountered during execution; and (c) the basic procedure followed by the controller in decoding the instructions and supervising their execution throughout the STEP machine.

1.1 Step lists

The "program" that the controller executes consists of a sequence of instructions, called *step list*, residing in the host's memory.[1] Typically, the step list and the associated data buffers will have been constructed at leisure by a user program running under UNIX, without much concern for real-time performance. On the other hand, dependable real-time performance is expected of the controller; to this end, the controller is given *bus master* access to the SBus when interpreting the step list.

The data buffers provide designated areas for transfering data to and from the STEP array. A typical data buffer would be a LUT buffer, i.e., a memory image of a STEP lookup table.

The step lists and the associated data buffers reside in an area of host memory that is shared beween the host and the controller. Several step lists (with the associated sets of data buffers) may be present in shared memory at the same time, ready to be executed in turn. In the SPARCstation, this shared memory is mapped simultaneously into the kernel's virtual address space (where it is seen by the controller) and into the user's address space (where it is seen by the user's program). All shared memory is allocated in multiples of four 32-bit words (16 bytes).

[1] The step list is analogous to the "display list" routinely used for interfacing a conventional processor to a specialized, high-performance graphics processor.

1.1.1 Instruction format

Each step-list instruction consists of exactly four 32-bit words named as follows

$$\boxed{\begin{array}{l}\text{opcode}\\ \text{buffer pointer}\\ \text{register length}\\ \text{link to next instruction}\end{array}}\,. \tag{1.1}$$

Typically, an instruction makes reference to some associated data. These data can be in an *inline buffer*, immediately following the instruction, or in a separately allocated buffer. In either case, the buffer pointer field in (1.1) points to the origin of this buffer. Small amounts of data can also be encoded in *immediate* form, i.e., within the instruction itself; in this case no data buffer is used.

Scanning of a step list proceeds sequentially from the beginning to the end of the list—there are no flow-control instructions such as IFs, JUMPs, or GOTOs.

1.1.2 Instruction fetching

Writing a memory address to the Next List Pointer tells the controller that a step list is present in memory at the given address and that the controller should start executing it as soon as it is done with the list (if any) which it is currently executing. The exact sequence of events will depend on the current state of the controller. For the purposes of writes to the NLP, the controller can be in one of three states: *execute*, *wait-for-host*, and *error-wait-for-host*.

In the execute state, the controller is still executing the previous step list. Assuming that no errors are encountered, the controller will start executing the new list as soon as the previous one is finished.

In the wait-for-host state, the controller is idle. As soon as the NLP is written to, the controller immediately starts execution of the given list.

In the error-wait-for-host state, besides writing to the NLP it is necessary to indicate—through the "enable-dvma" option—to resume execution; simply writing to the NLP is not sufficient. The reason for this is to prevent execution from continuing when the interface receives an error during the write to the NLP, in which case execution of the new step list should not commence.

1.2 STEP controller

As explained in the *Architecture Reference*, the STEP controller's main function is to interpret step lists, which reside in the host's memory, traslating them into sequences of transactions between data buffers in host memory and STEPchip registers in the modules.

The controller contains a number of hardware registers, called *control registers*, which are use to initialize, direct, and monitor its operation. These registers, some parts of which are read-only, some write-only, and others read/write, are memory mapped on the SBus as four consecutive 32-bit words which are accessed as SBus slaves.

Specifically, the registers comprise one read/write register (the Next List Pointer), three read-only registers, and three write-only registers. Read-only and write-only registers share the same three memory locations. Thus, when read from, the controller's control registers have the following description

controller's registers (read)	
NLP	Next List Pointer
ISR	Interrupt Status Register
CIP	Current Instruction Pointer
PIP	Previous Instruction Pointer

, (1.2)

and when written to they have the following description

controller's registers (write)	
NLP	Next List Pointer
ICR	Interrupt Control Register
DSL	Display Scan Length
DBL	Display Blank Length

. (1.3)

In the three pointers above (NLP, CIP, and PIP), the low four bits have a special use. Since the addresses to which the pointers refer are always 16-byte aligned, the pointers' low four bits are predictably 0s and need not be explicitly given. In their stead, those four bit positions are used as read-only windows on certain status bits (writes to the same bit positions are ignored.)

1.2.1 Next List Pointer

The Next List Pointer is a read/write register. Bits 4–31 (`next_list`) represent the starting address (without the low four bits, which are understood to be 0s) of the step list to be executed next. Bits 0–3 are read-only status bits with the following meaning

Next List Pointer status bits		
bit	name	function
0	`nlp_written`	the controller is still executing the previous display list, and has not yet made use of the NLP.
1	`exception_status`	the controller has seen an exception (timeout or STEP interrupt).
2	`wait_for_host`	the controller is in "wait-for-host" mode.
3	`interface_halted`	the controller is currently halted.

. (1.4)

[As soon as the controller is free (it might have been executing another step list), it looks at the NLP and starts executing the step list pointed to by it. At the same time, it signals to the host (by means of an `nlp_int` interrupt) that the current step-list pointer value has been read from the NLP register, and a new value can be written in it.] ♠

[revise this paragraph] Bit 3 is a status bit which is set if CAM is currently scanning. Bit 2 is set if the interface is in the "DMA Active" mode, meaning that it is performing, or ready to perform the execution of step list. Bit 1 is set if the interface is in "wait-for-host" mode, and bit 0 is set if the current NLP has not yet been processed by the interface (meaning that the interface is still working on the previous display list.) ♠

1.2.2 Interrupt Status/Control Registers

The Interrupt Status Register (Fig. 1.1) and the Interrupt Control Register (Fig. 1.2) are used to monitor, enable, or disable interrupts generated by the controller. The latter register is is also used to reset, halt, and restart the controller, and to reset the STEP array.

The controller can place on the SBus five kinds of interrupts, namely,

controller interrupts	
name	description
`soft_int`	Software interrupt.
`cam_int`	STEP-originated interrupt.
`sbus_int`	The controller detected an SBus error.
`timeout_int`	Timeout in communicating with the STEP array.
`nlp_int`	Next List Pointer interrupt: the controller has started executing the step list indicated by the Next List Pointer; a new step list can be scheduled by writing a new address to the Next List Pointer.

(1.5)

To each of these interrupts the controller associates an *interrupt enable* bit and an *interrupt flag* bit, whose functions will be explained in a moment. The five enable bits and the five flag bits are directly readable respectively in positions 0–4 and 8–12 of the Interrupt Status Register.

The five flag bits cannot be directly written by the host. Each of these bits is set by the occurrence of the corresponding interrupt condition (e.g., timeout interrupt); they are collectively cleared by any read operation performed on the Interrupt Status Register.

Whether a flagged interrupt actually generates an interrupt on the SBus depends on whether that particular interrrupt is enabled. The five enable bits of the Interrupt Status Register are not writable in a conventional way. They are *set* by writing a 1 to respectively bits 8, 9, 10, 11, and 12 of the Interrupt Control Register, and *cleared* by writing a 1 to, respectively, bits 0, 1, 2, 3, and 4 of the same register.

The occurence of a `cam_int` or a `timeout_int` suspends the list scanning activity of the controller, regardless of whether these interrupts are enabled (and thus reported to the host as SBus interrupts). However, these interrupts can be masked at an earlier stage, so that they will not be seen at all by the controller. This is done by two more enable bits, namely `cam_exception` and `timeout_exception`, which can be set or cleared in a way similar to the other enable bits, by writing a 1 to, respectively, bits 5 and 6 or bits 13 and 14 of the Interrupt Control Register.

Finally, as shown in Fig. 1.2, four more control functions are performed by writing a 1 to bits 12, 13, 16, and 31 of the Interrupt Control Register.[2]

[2]Bits 12 and 13 also happen to affect exception control (see "also" entries in Fig. 1.2); this is an oversight—fortunately a harmless one—and will be fixed with the next release of the controller chip set.

Interrupt Status Register	
bit	name
0	soft_int_flag
1	cam_int_flag
2	sbus_int_flag
3	timeout_int_flag
4	nlp_int_flag
5–7	—
8	soft_int_enable
9	cam_int_enable
10	sbus_int_enable
11	timeout_int_enable
12	nlp_int_enable
13–31	—

Figure 1.1: Structure of the Interrupt Status Register. Bits 0–4 (interrupt flags) are set by the occurrence of the appropriate interrupt conditions, and are automatically cleared by any read of this register. Bits 8–11 are set by writing 1s to bits 8–11 of the Interrupt Control Register, and cleared by writing 1s to bits 0–5 of the same register.

Interrupt Control Register	
bit	effect of writing a 1
0	`soft_int` disable
1	`cam_int` disable
2	`sbus_int` disable
3	`timeout_int` disable
4	`nlp_int` disable
5	`cam_exception` disable
6	`timeout_exception` disable
7	—
8	`soft_int` enable
9	`cam_int` enable
10	`sbus_int` enable
11	`timeout_int` enable
12	`nlp_int` enable
	also: Clear fault
13	`cam_exception` enable
	also: Clear exception
14	`timeout_exception` enable
16	Halt the controller.
17–30	—
31	Reset the controller.

Figure 1.2: Structure of the Interrupt Control Register.

1.2.3 Current Instruction Pointer

The Current Instruction Pointer is a read-only register (writes to this location affect the Display Scan Length; §1.2.5). Bits 4–31 (`curr_inst`) represent the address (without the low four bits, which are understood to be

0s) of the step-list instruction currently being executed. Bits 0–3 are read-only status bits (cf. §1.2) with the following meaning

Current Instruction Pointer status bits		
bit	name	function
0	`sbus_error`	The controller is suspended because of SBus error.
1	`partial_write`	Owing to an SBus error, only a partial write was performed by the controller.
2	`done_status`	controller is not relaying to STEP.
3	—	Reserved.

(1.6)

♠ [Edit bit 2.]

1.2.4 Previous Instruction Pointer

The Previous Instruction Pointer is a read-only register (writes to this location affect the Display Blank Length; §1.2.5). Bits 4–31 (`prev_inst`) represent the address (without the low four bits, which are understood to be 0s) of the step-list instruction executed before the current one. Bits 0–3 are read-only status bits (cf. §1.2) with the following meaning

Previous Instruction Pointer status bits		
bit	name	function
0	`cam_int_active`	This bit reflects the status of the Interrupt signal of the CBus, resynchronized to the SBus clock. It allows the host to directly see whether there is an interrupt on the CBus.
1	`cam_scanning`	STEP is scanning
2		(Reserved for SPARCstation-2 bit.)
3		(Reserved.)

♠ (1.7)

Bit 2 (?) of the previous instruction register indicates if the long-burst transfer mode supported on the SPARCstation-2 is being used by the interface board.

1.2.5 Display Scan/Blank Length

1.2.6 Remarks

♠ [understand and drop in place the following remarks.]

wait-for-host is fixed by writing to the NLP

all error interrupts except new-list (NLP) and soft cause dvma-inactive and wait-for-host mode.

cam-error: parity and verify-write errors are not signalled until end of transfer, this interrupt will always be in the "next" instruction.

current-instruction: generally this is the current instruction if "stop" is received. This instruction should have been attempted, the previous instruction has generally been completed.

Check the state that interface and cam are in when a hardware interrupt is received. do not assume that interface is in some "clean" state.

A Controller chip schematics

This appendix contains the complete schematics for the three controller chips. The nodes of the schematics hierarchy are called **modules**.[1]

Most modules are in the form of *circuit* schematics; the other modules, having the tag '`_p`' at the end of their names, are *state-machine* tables.

The three hierarchy trees, headed by the nodes **`ctlsvchip`**, **`dataregs`**, and **`fsmchipand`**, expand as follows[2]

```
ctlsvchip
    clrsync
    control
        controlreg
            upctr16
                upctr4_a
        dnct_ctl_1
        dnctr16_a
            dnctr4_a
            dnctr4_b
        opcode_reg
    csvsel
    slave
        eosi_dcd
        hnp_ctl
            hnp_fsm_p
        reg_1
            cmd_reg
            int_reg
                int_cond
                    ci_sync
        slavereg
            bus16_2
            ptr_reg
            sb_va_gen
            sync_gen
                sync_ctr
                    dnctr8
                        dnctr4
                sync_encode
                sync_fsm_p
                xsync
        wait_stat
            xsync
        xsync
dataregs
    clrsync
    datain
        din_ctl
        din_sel
            upctr2_a
        dinreg
        ptygen8
            pty_p
    dataout
        dout_sel
            upctr2_a
        doutreg
        ldsel
    fsm_dec_p
    sb_drv
    sv_oncircuit
fsmchip
    clrsync
    dout_stat
        ps_sync
    drdsync
    eos_dec
    fsm_enc_p
    pbus_ctl
        davsync
        donep_stat
        pb_ctl
            pctlfsm_p
        pb_rd
            prdfsm_p
        pb_wr
            pwrfsm_p
        s_psyn
        tmctr
            upctr8
                upctr4
            xsync
        upctr2
        xsync
    ps_sync
    sbus_ctl
        donesync
        errorgen
        ifc_rst
            timer4
        sctlfsm_p
        sedge
        srdfsm_p
        swrfsm_p
        upctr3
    sv_sbug
        ack_reg
        sbugfixfsm_p
        svfsm_p
    wr_dav
        davsel
            davsync
```

[1] These are Valid design modules, not to be confused with the physical modules that make up a STEP system.

[2] Two modules, namely **`weight`** and **`mff`**, define very low-level primitives and have been omitted.

A.1 List of controller chip schematics

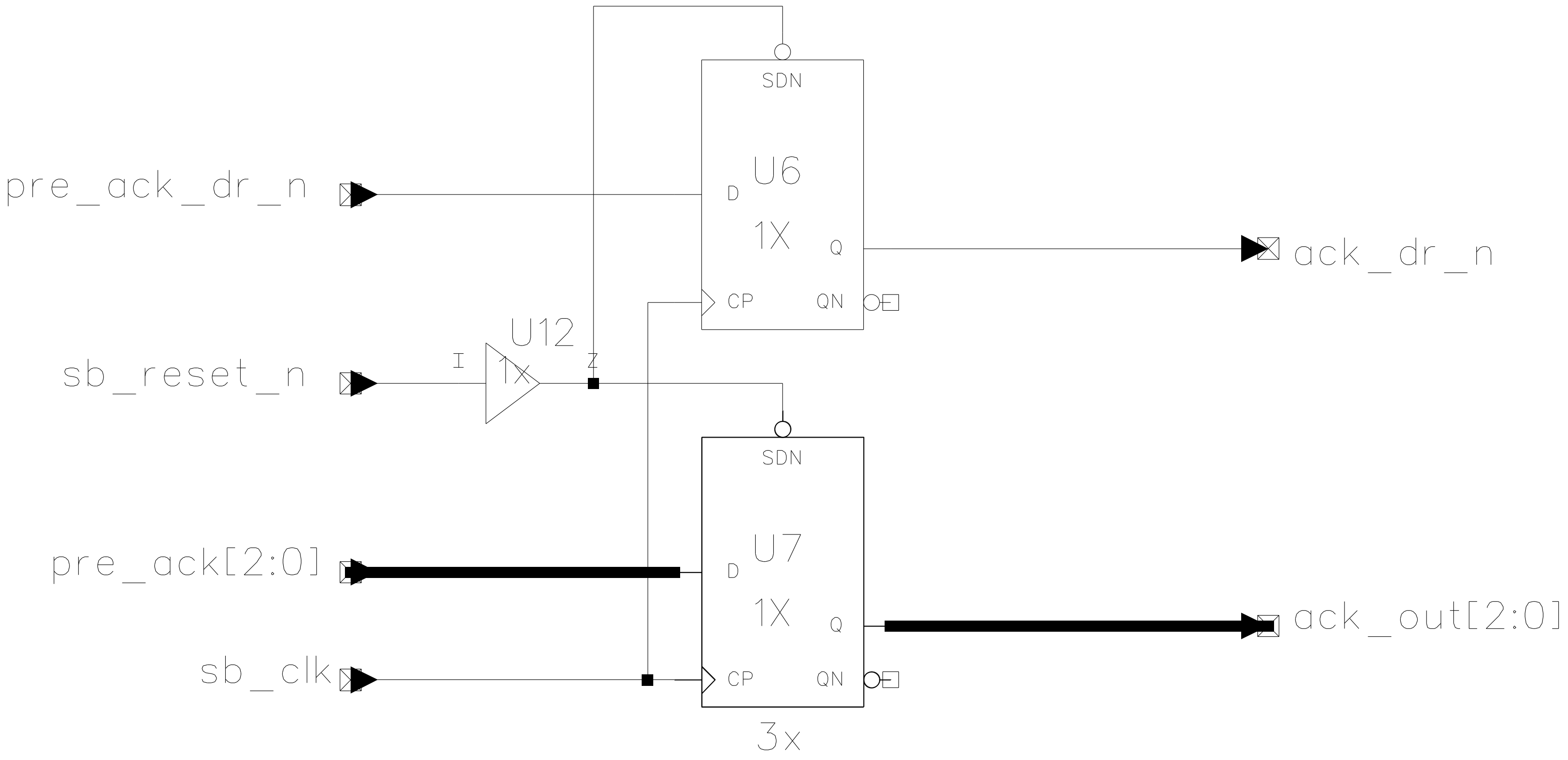
The modules documented in the following pages appear in the order listed below. Circuit schematics are given as PostScript files with extension `ps`. State-machine tables are given as text files with extension `sm`.

Thus, for example, module `dataregs` is documented by the schematics `dataregs.ps`, while module `pty_p` is documented by the state machine `pty.sm`.

Controller chips

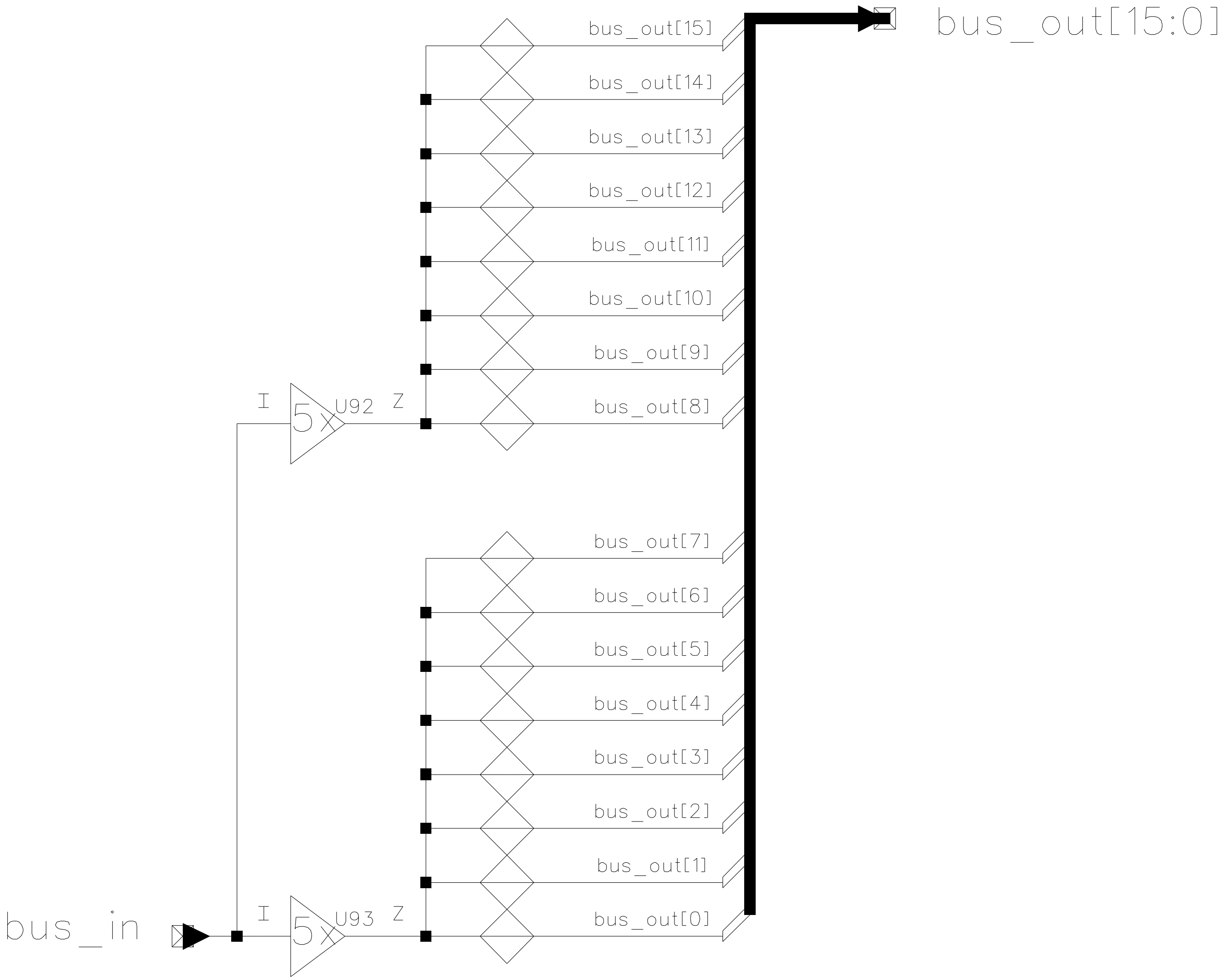
ack_reg.ps
bus16_2.ps
ci_sync.ps
clrsync.ps
cmd_reg.ps
control.ps
controlreg.ps
csvsel.ps
ctlsvchip.ps
datain.ps
dataout.ps
dataregs.ps
davsel.ps
davsync.ps
din_ctl.ps
din_sel.ps
dinreg.ps
dnct_ctl_1.ps
dnctr16_a.ps
dnctr4.ps
dnctr4_a.ps
dnctr4_b.ps
dnctr8.ps
donep_stat.ps
donesync.ps
dout_sel.ps
dout_stat.ps
doutreg.ps
drdsync.ps
eos_dec.ps
eosi_dcd.ps
errorgen.ps
fsm_dec.sm
fsm_enc.sm
fsmchip.ps
hnp_ctl.ps
hnp_fsm.sm
ifc_rst.ps
int_cond.ps
int_reg.ps
ldsel.ps
opcode_reg.ps
pb_ctl.ps
pb_rd.ps
pb_wr.ps
pbus_ctl.ps
pctlfsm.sm
prdfsm.sm
ps_sync.ps
ptr_reg.ps
pty.sm
ptygen8.ps
pwrfsm.sm
reg_1.ps
s_psyn.ps
sb_drv.ps
sb_va_gen.ps
sbugfixfsm.sm
sbus_ctl.ps
sctlfsm.sm
sedge.ps
slave.ps
slavereg.ps
srdfsm.sm
sv_oncircuit.ps
sv_sbug.ps
svfsm.sm
swrfsm.sm
sync_ctr.ps
sync_encode.ps
sync_fsm.sm
sync_gen.ps
timer4.ps
tmctr.ps
upctr16.ps
upctr2.ps
upctr2_a.ps
upctr3.ps
upctr4.ps
upctr4_a.ps
upctr8.ps
wait_stat.ps
wr_dav.ps
xsync.ps

[la]ack_reg by nhm on 16-Feb-93 at 12:18 A.M.



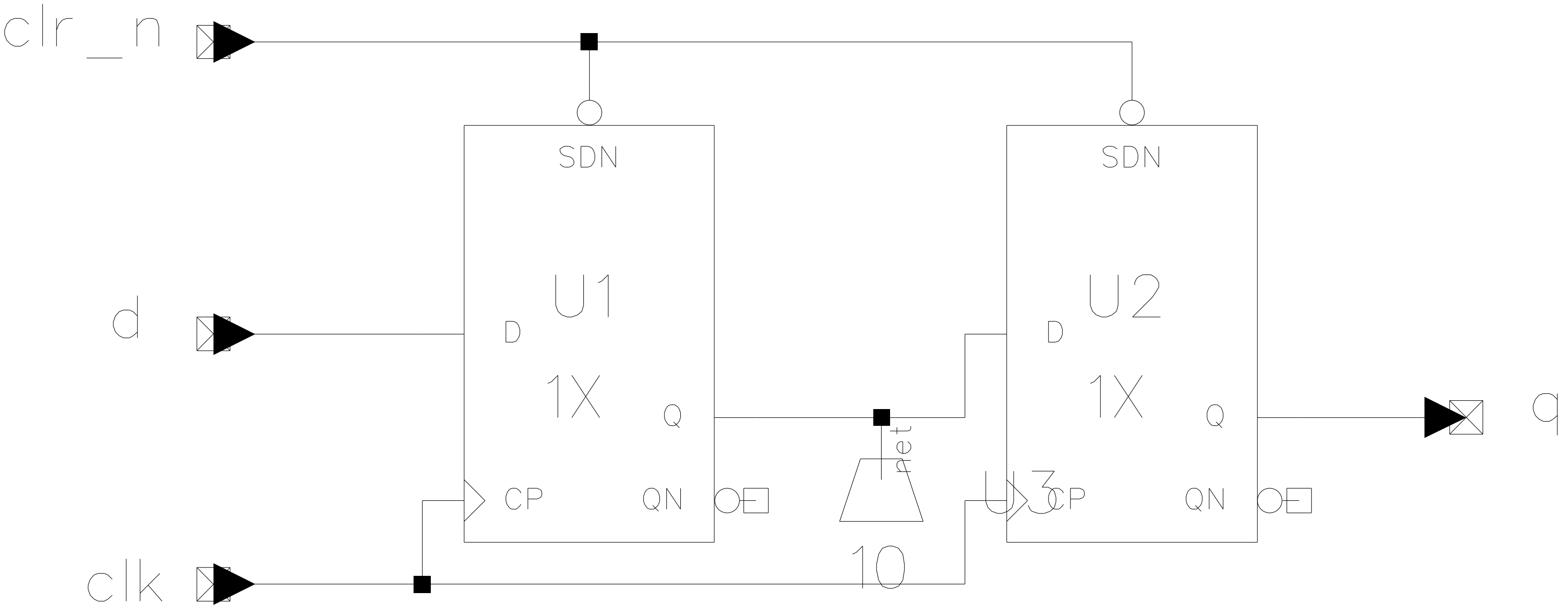
Controller chips

ack_reg.ps



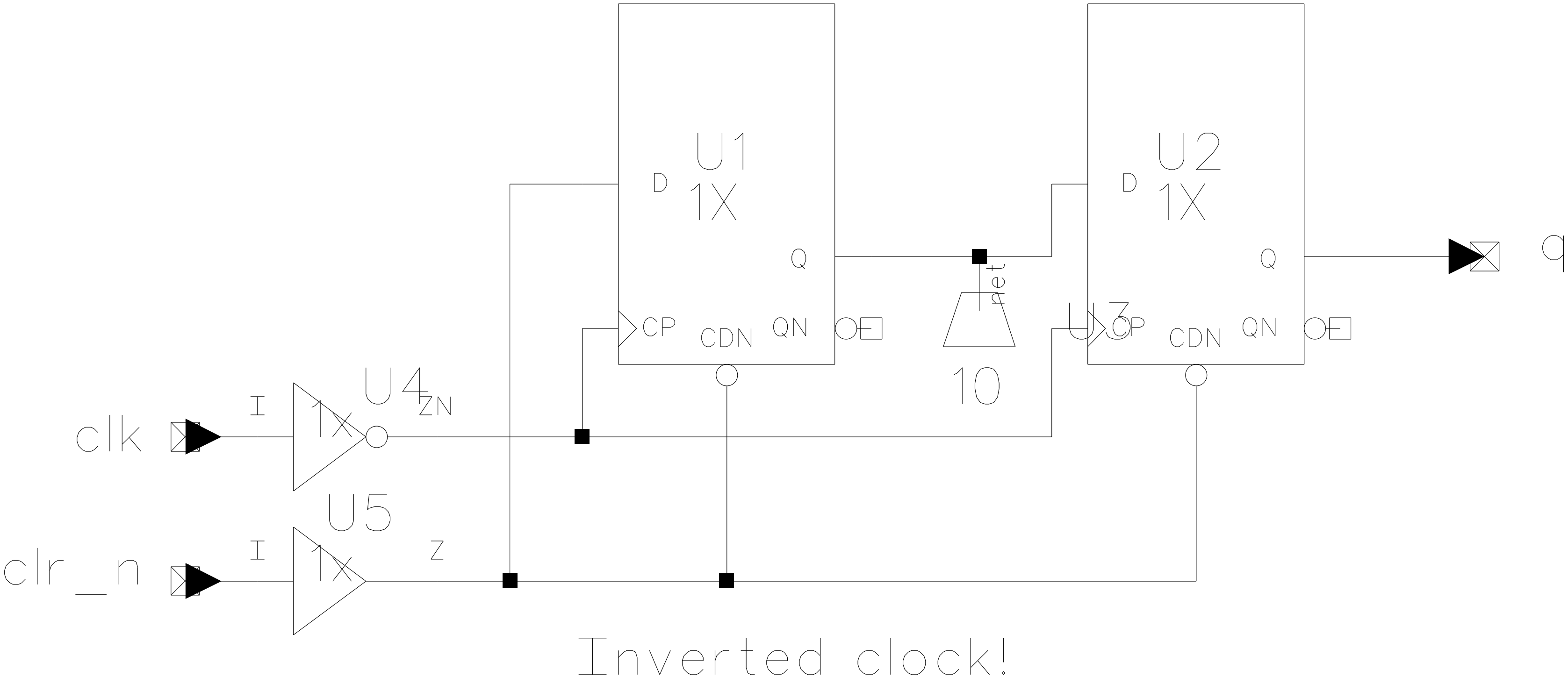
bus_out[15:0]
bus_out[15]
bus_out[14]
bus_out[13]
bus_out[12]
bus_out[11]
bus_out[10]
bus_out[9]
bus_out[8]
I
5x
U92
Z
bus_out[7]
bus_out[6]
bus_out[5]
bus_out[4]
bus_out[3]
bus_out[2]
bus_out[1]
bus_out[0]
bus_in
I
5x
U93
Z

[la]ci_sync by nhm on 16-Feb-93 at 12:18 A.M.

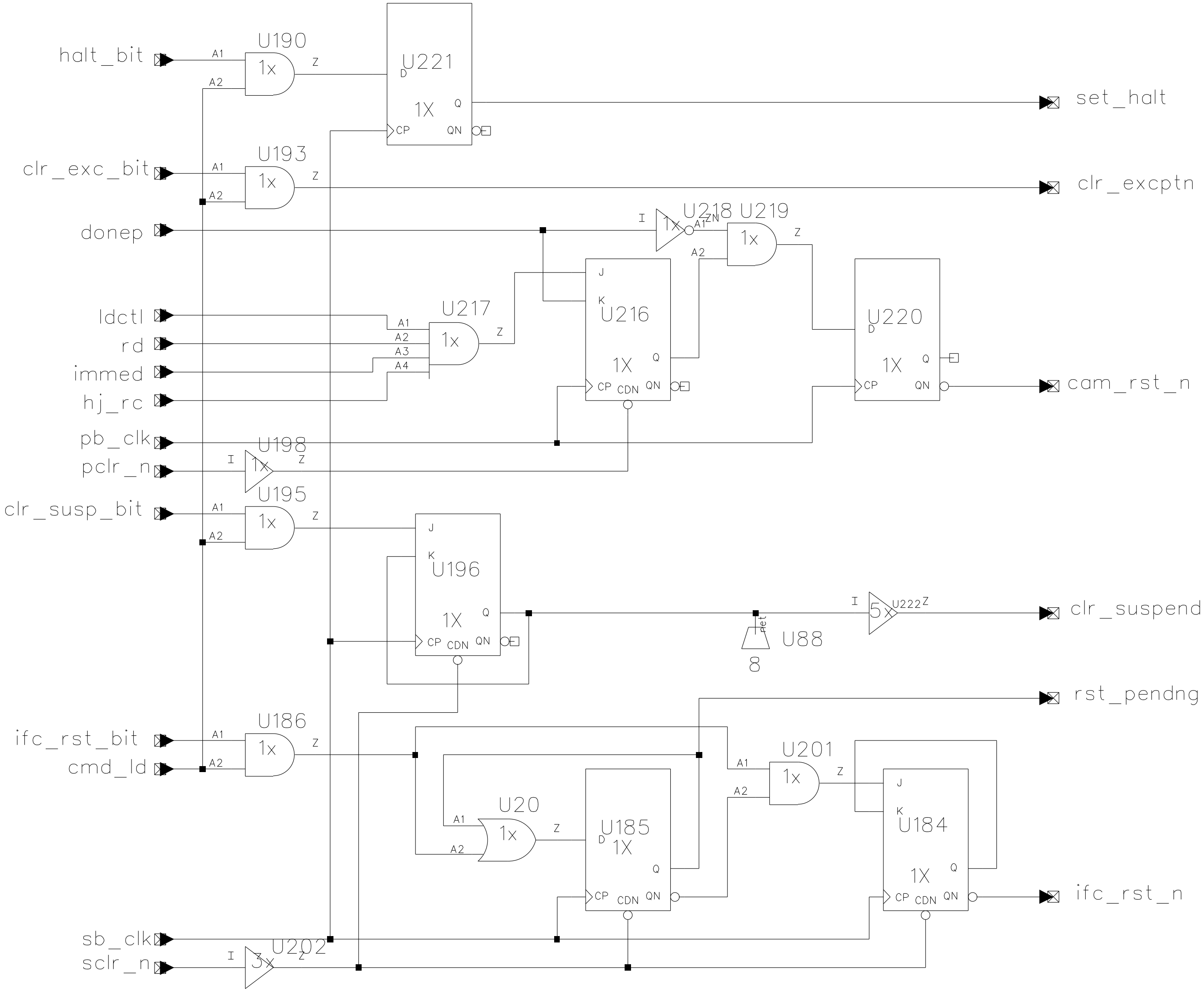


Controller chips

ci_sync.ps

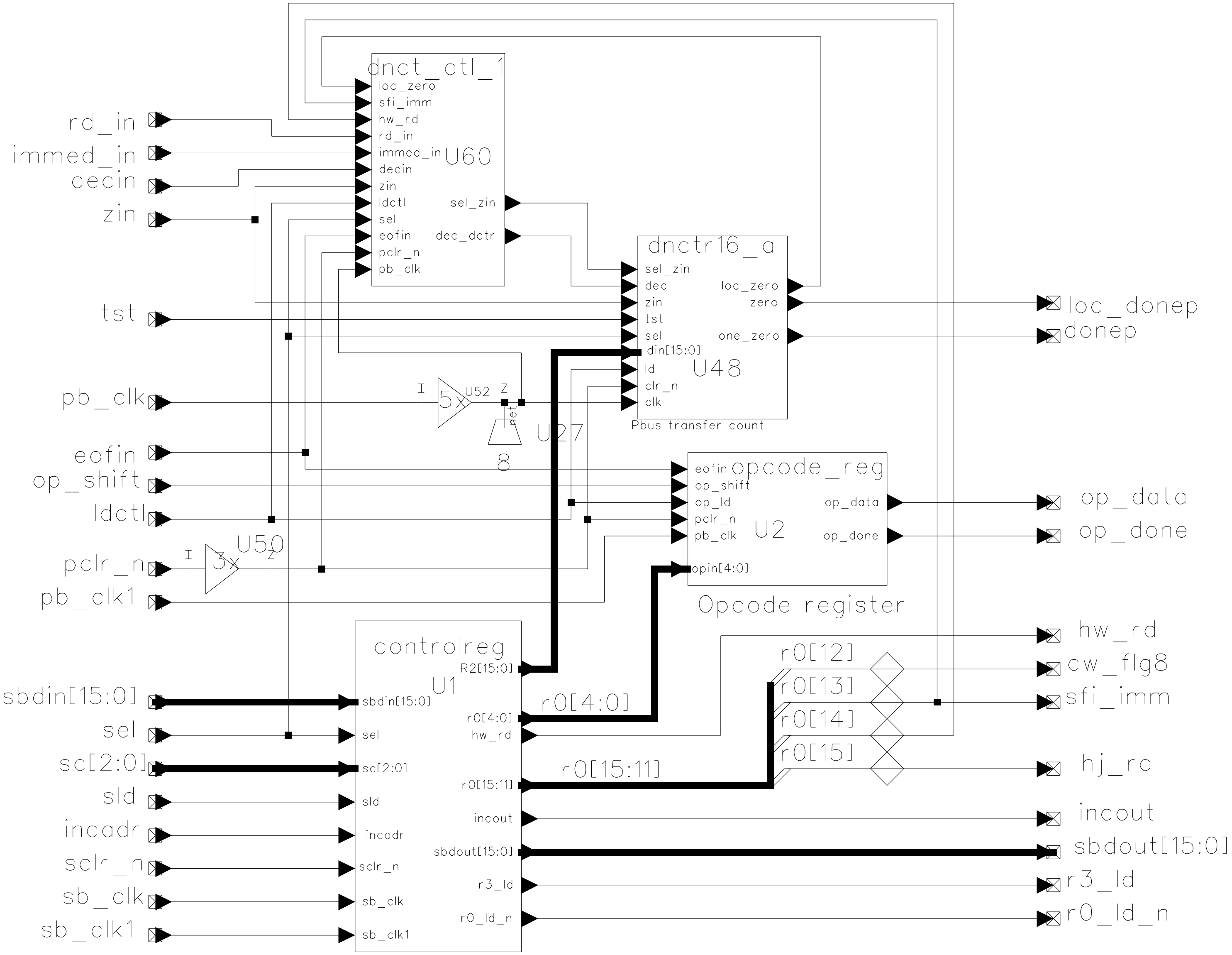

U1
1X
D
Q
CP
CDN
QN
U2
1X
D
Q
CP
CDN
QN
q
net
U3
10
clk
I
1x
U4
ZN
clr_n
I
1x
U5
Z
Inverted clock!

[la]cmd_reg by nhm on 16-Feb-93 at 12:18 A.M.



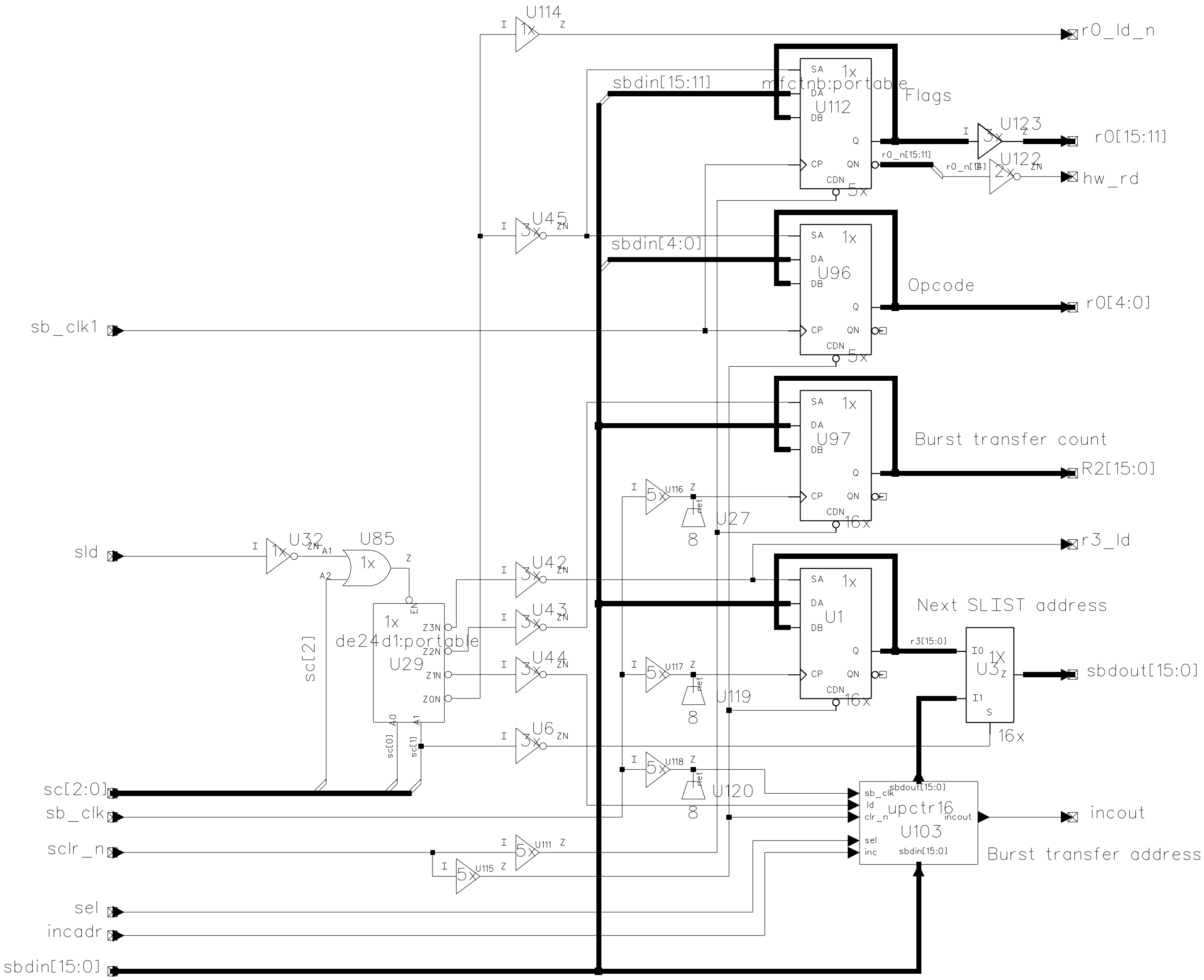
Controller chips

cmd_reg.ps



Controller chips

[la]controlreg by nhm on 16-Feb-93 at 12:18 A.M.

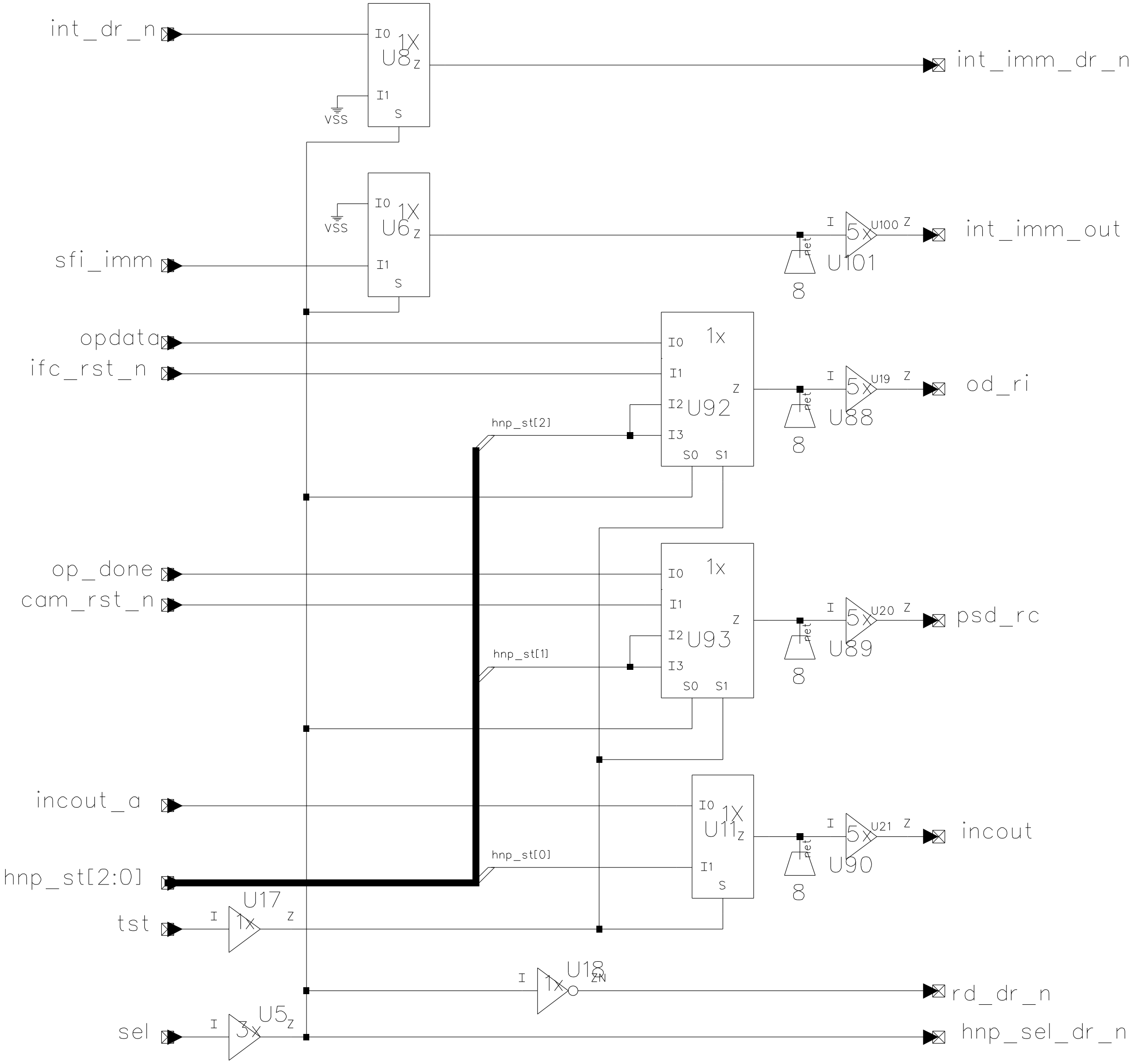


Controller chips

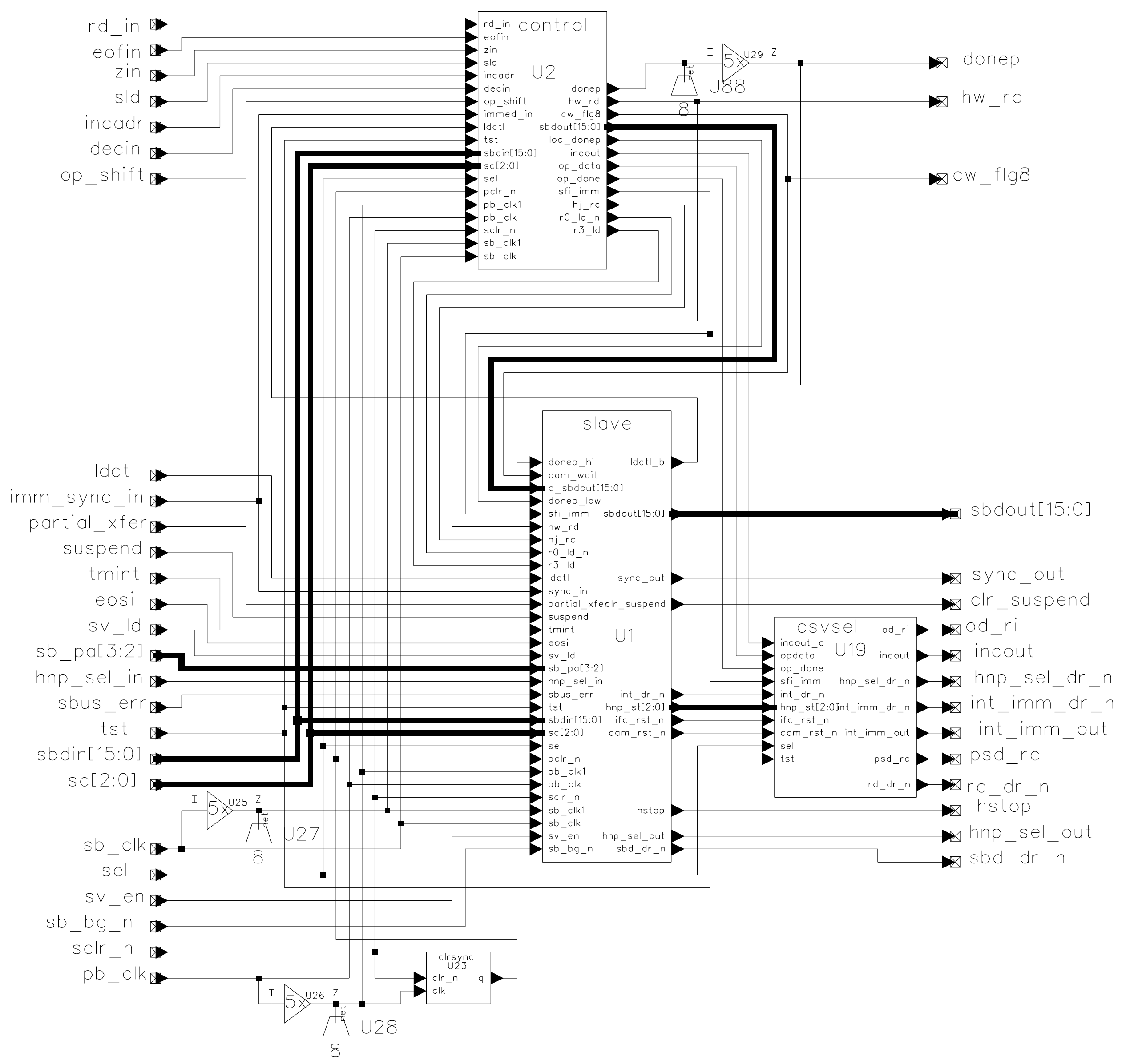
controlreg.ps

[la]csvsel by nhm on 16-Feb-93 at 12:18 A.M.

csvsel.ps



Controller chips

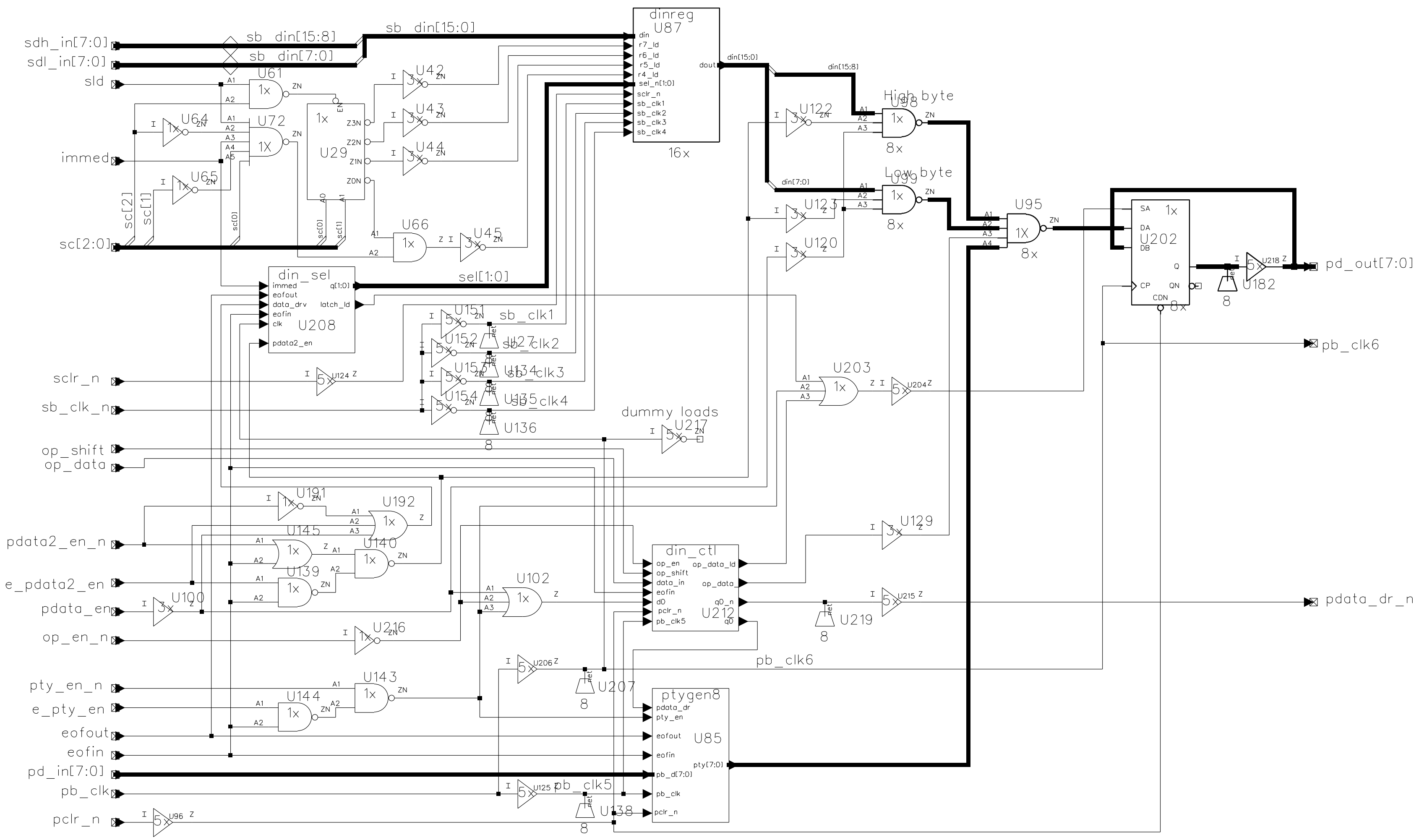


Controller chips

ctlsvchip.ps

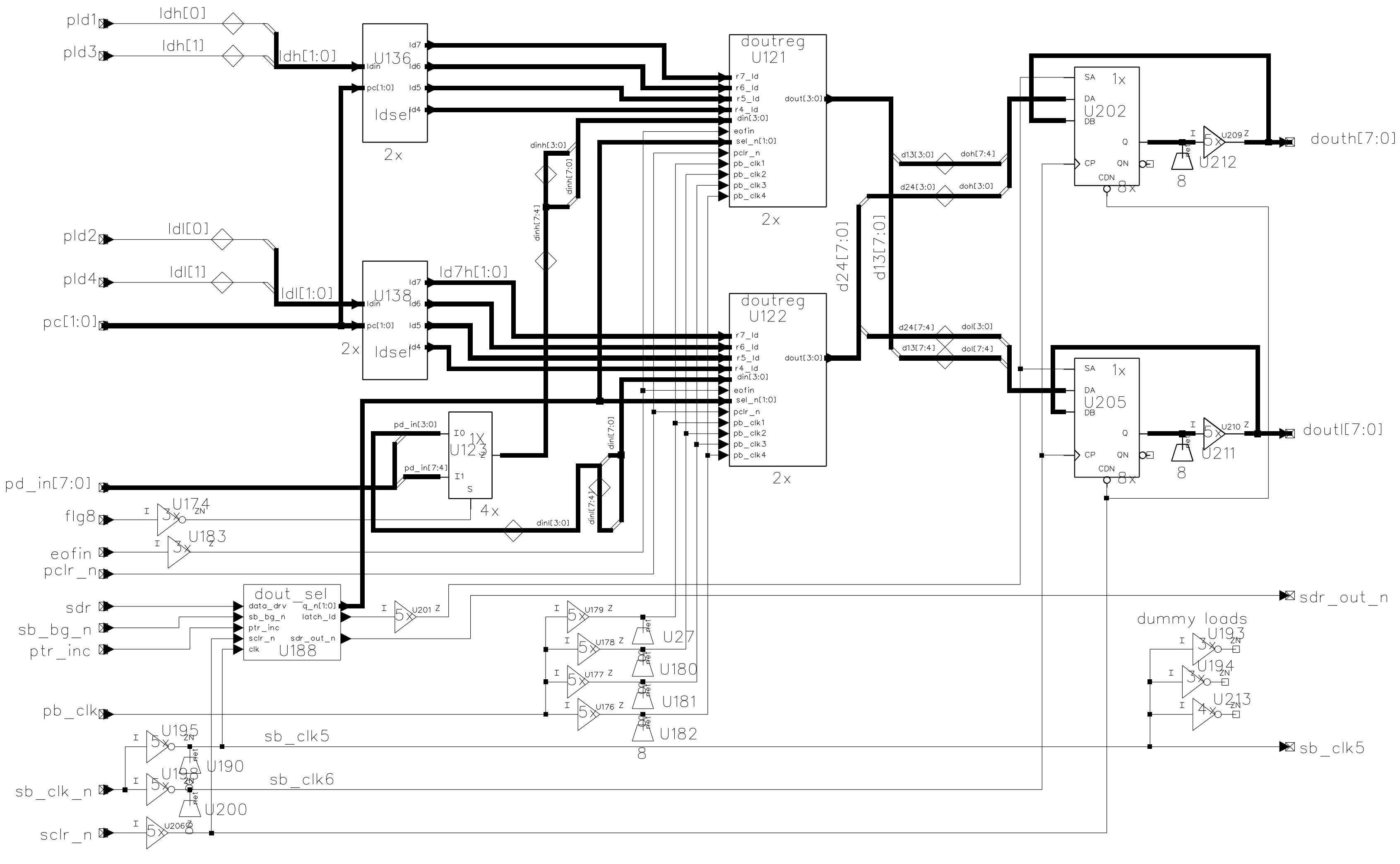
[la]datain by nhm on 16-Feb-93 at 12:18 A.M.

datain.ps



Controller chips

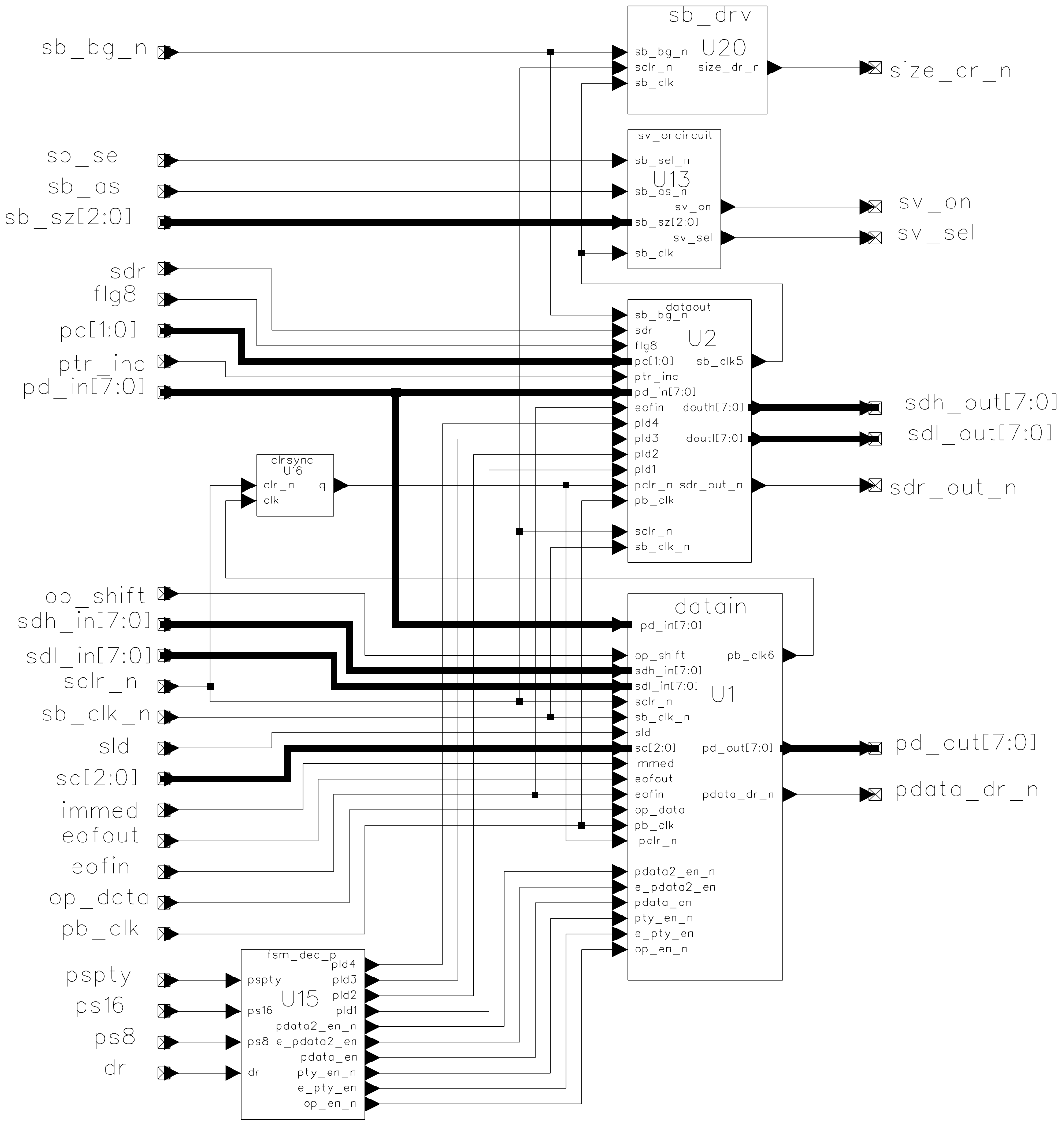
[la]dataout by nhm on 16-Feb-93 at 12:19 A.M.



Controller chips

dataout.ps

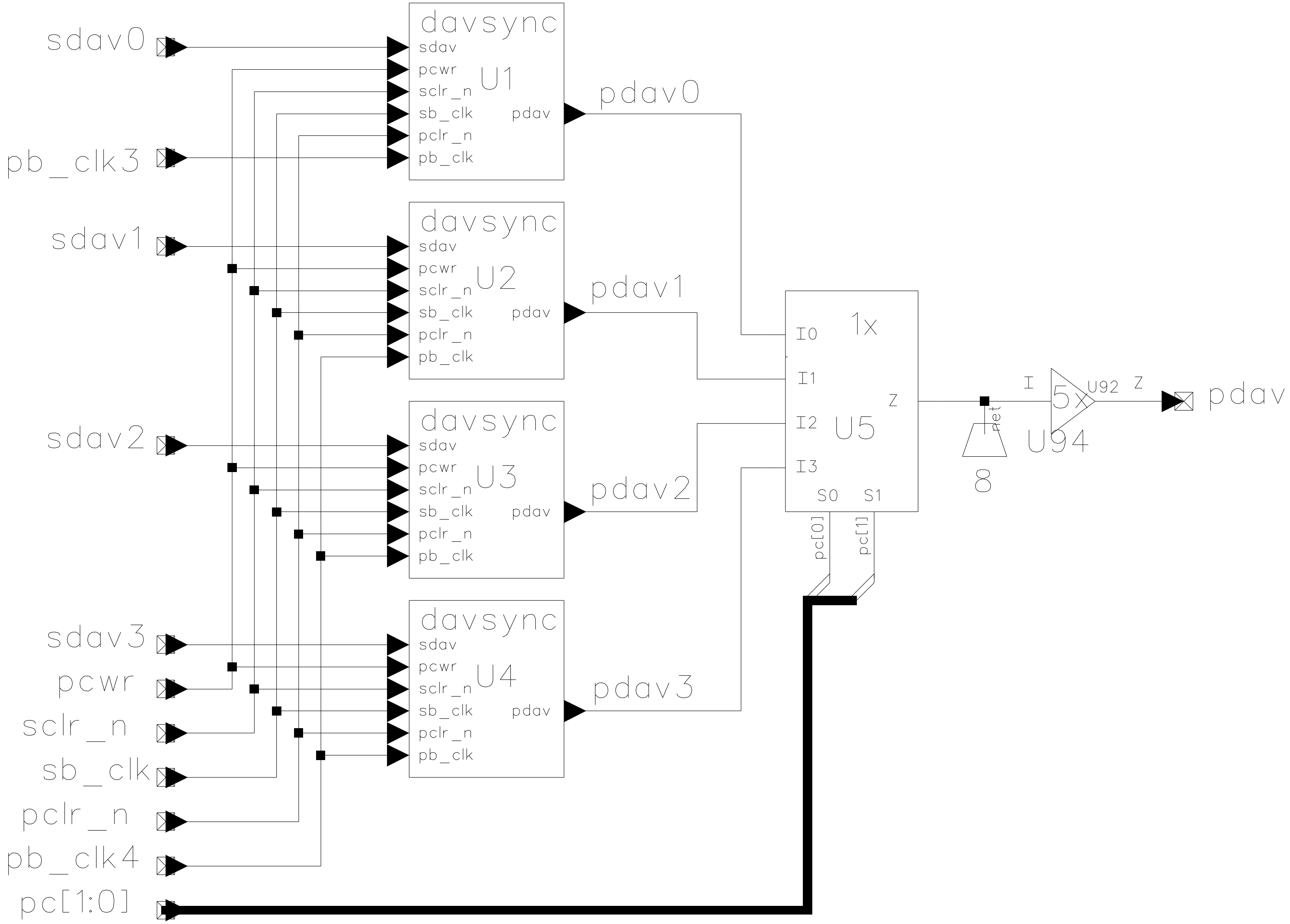
[la]dataregs by nhm on 16-Feb-93 at 12:19 A.M.



dataregs.ps

Controller chips

[la]davsel by nhm on 16-Feb-93 at 12:19 A.M.

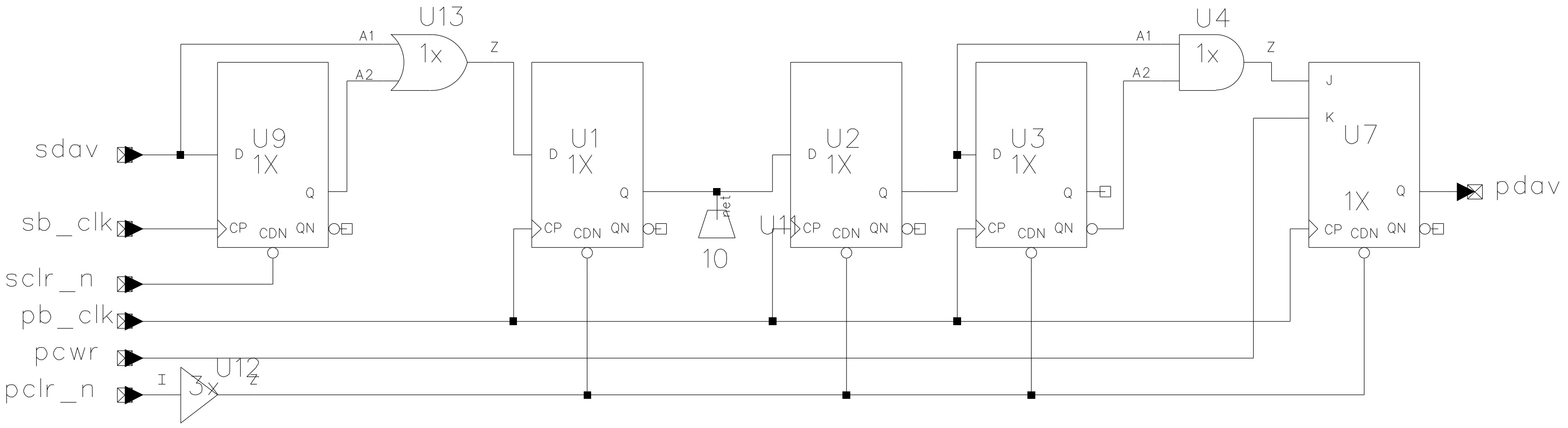


Controller chips

davsel.ps

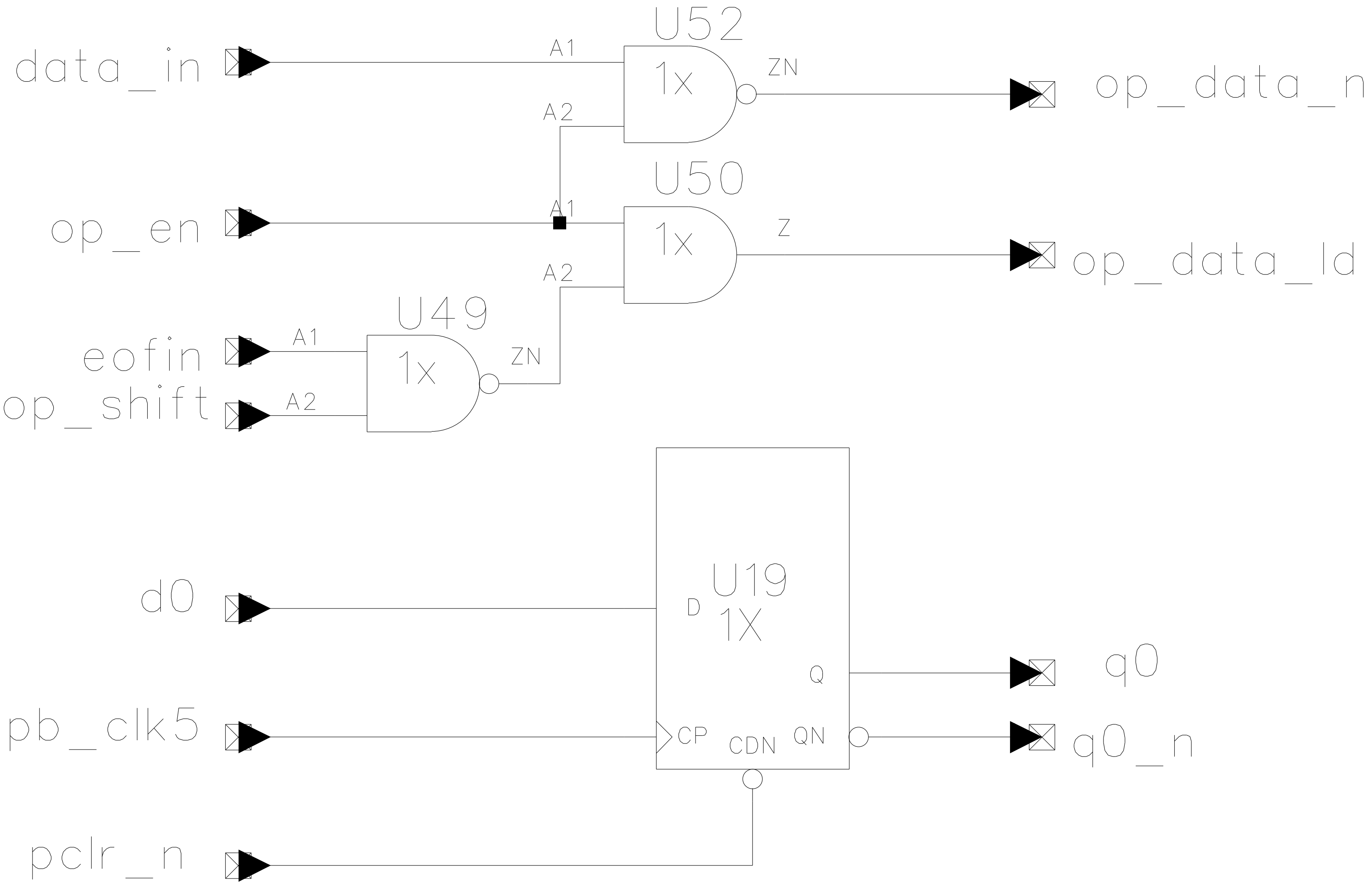
[la]davsync by nhm on 16-Feb-93 at 12:19 A.M.

davsync.ps



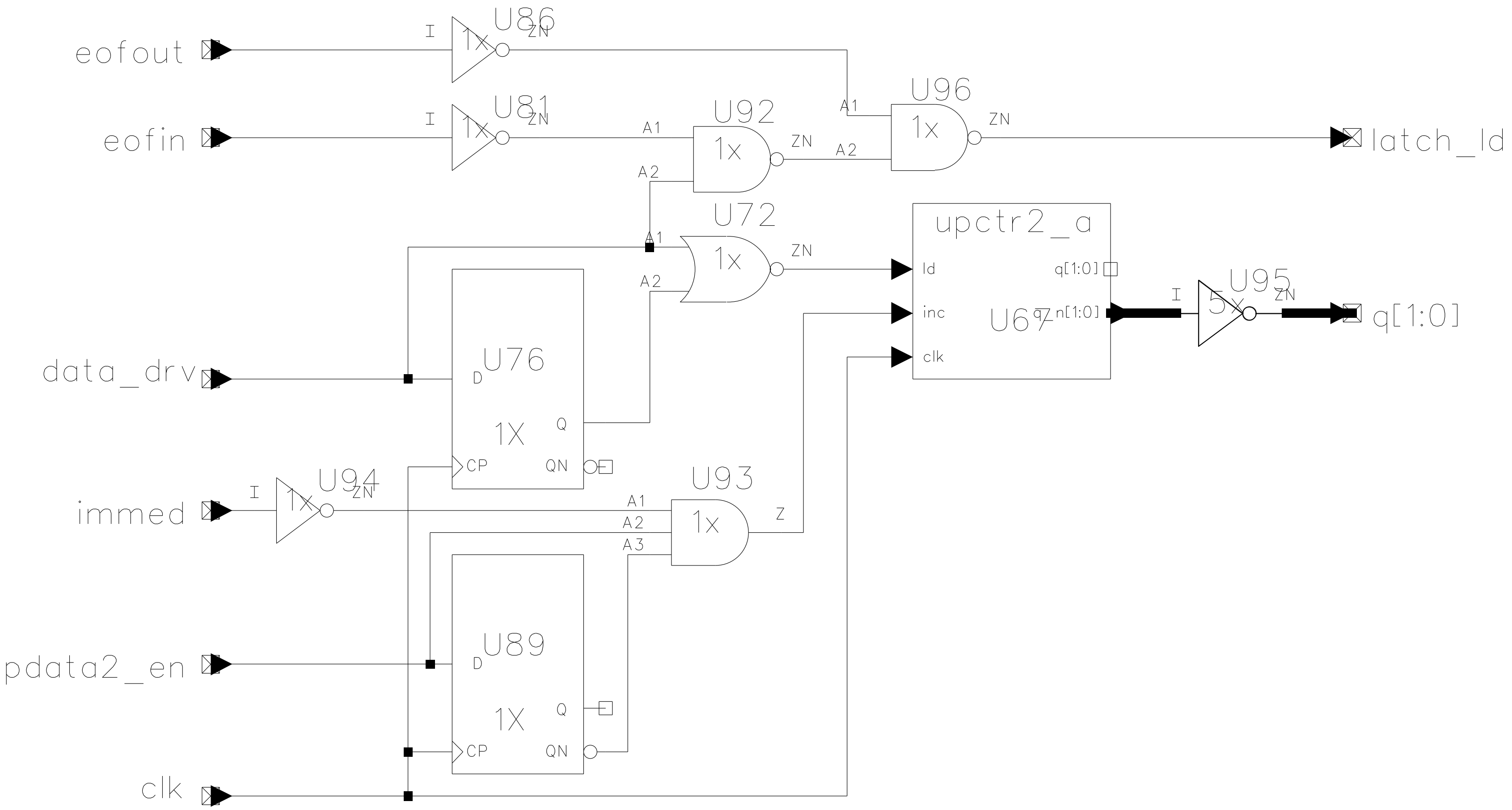
Controller chips

[la]din_ctl by nhm on 16-Feb-93 at 12:19 A.M.

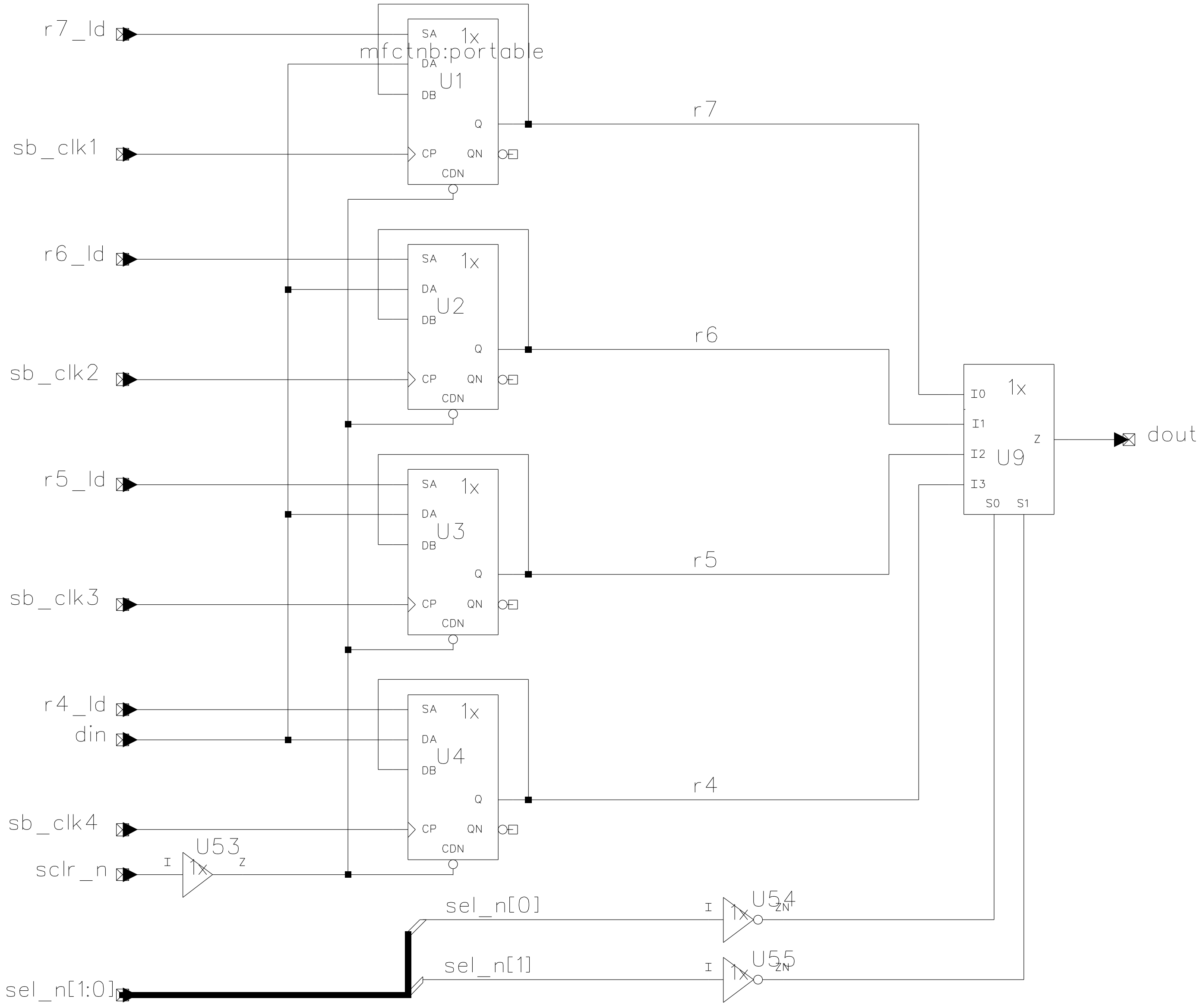


Controller chips

din_ctl.ps



Controller chips

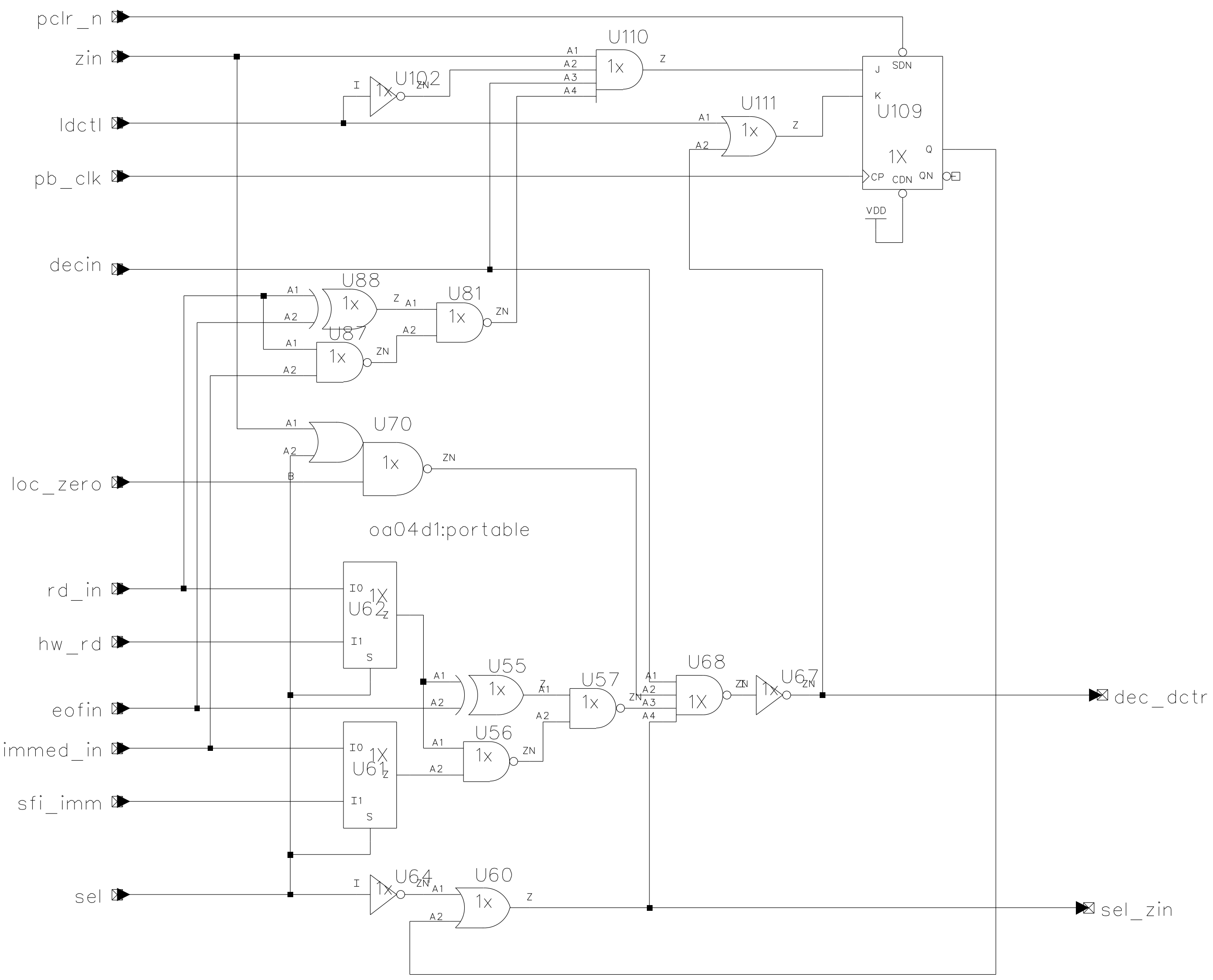


Controller chips

dinreg.ps

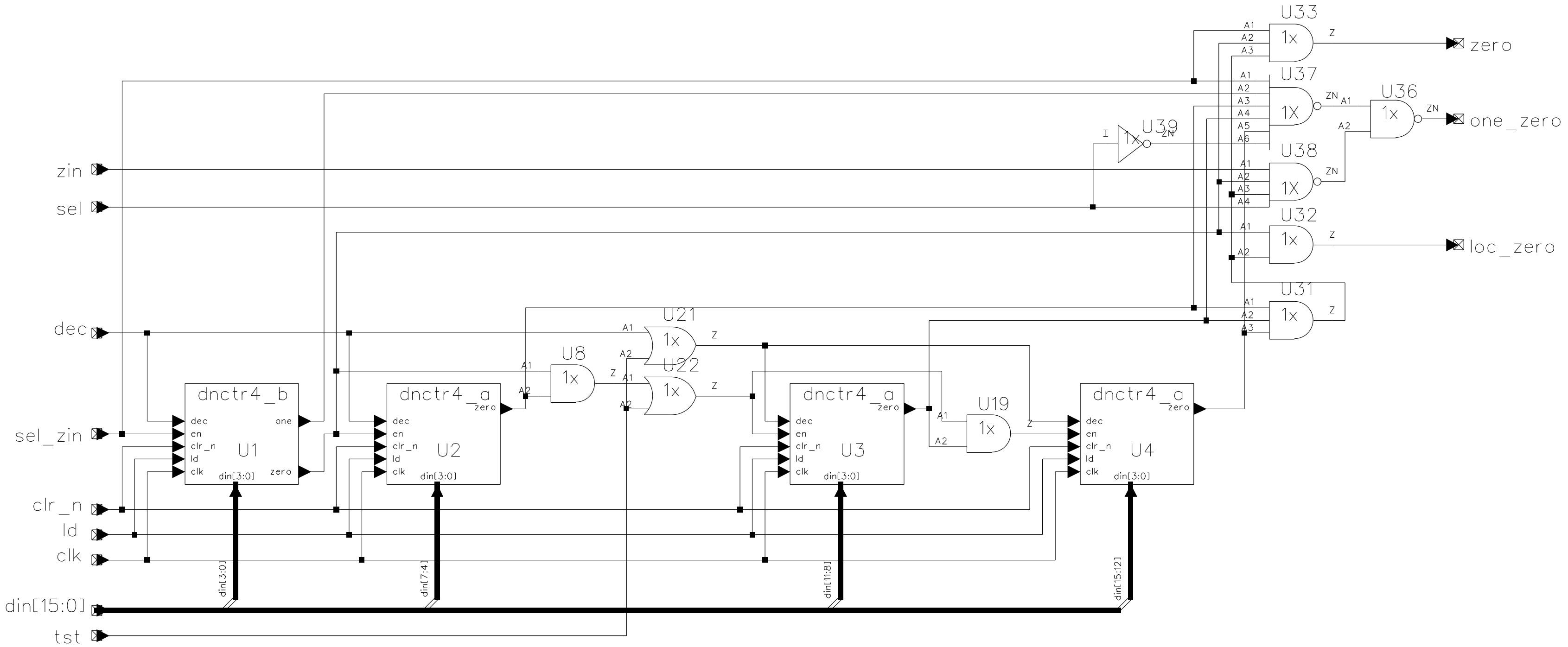
[la]dnct_ctl_1 by nhm on 16-Feb-93 at 12:19 A.M.

dnct_ctl_1.ps



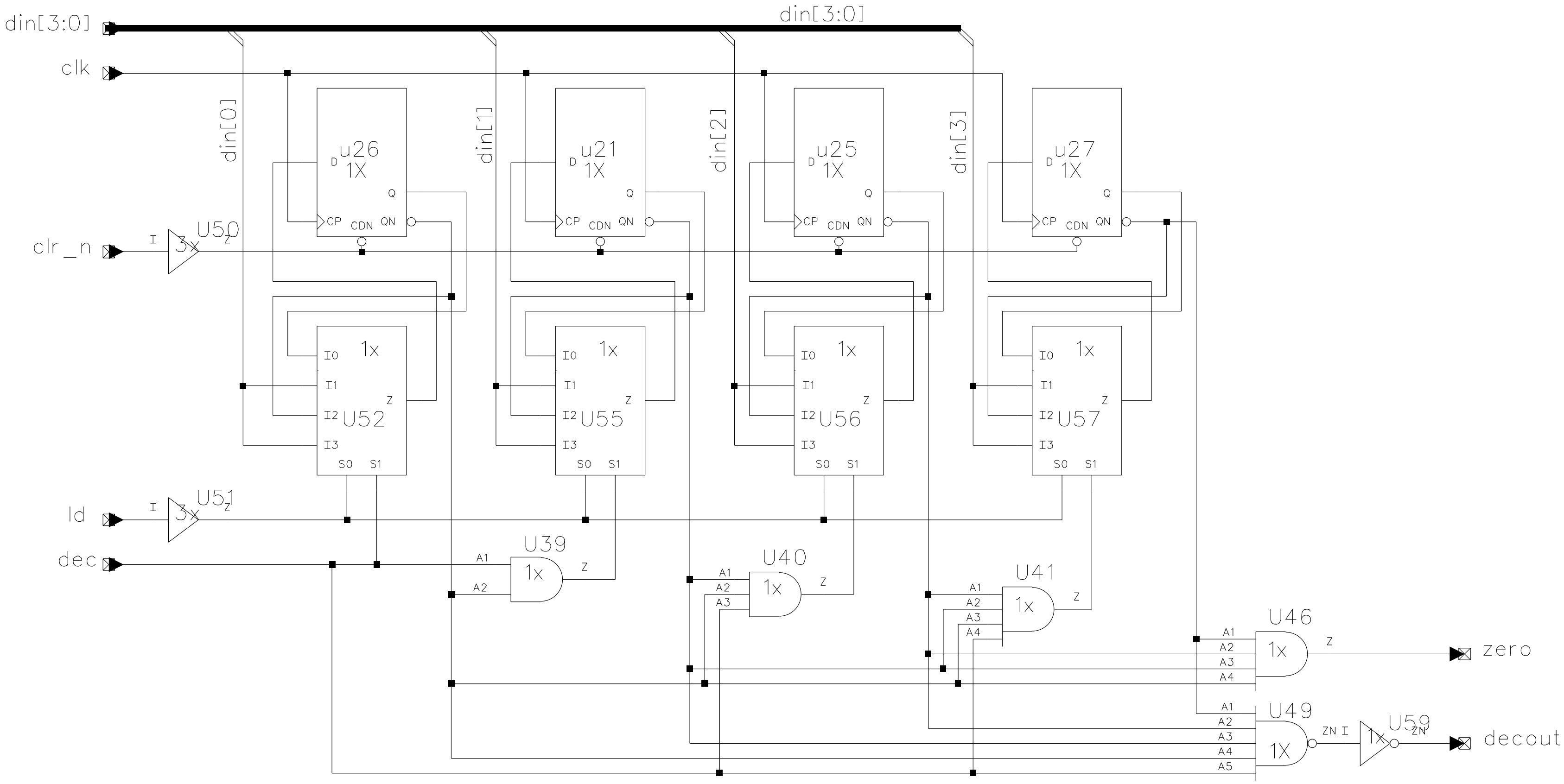
Controller chips

[la]dnctr16_a by nhm on 16-Feb-93 at 12:19 A.M.



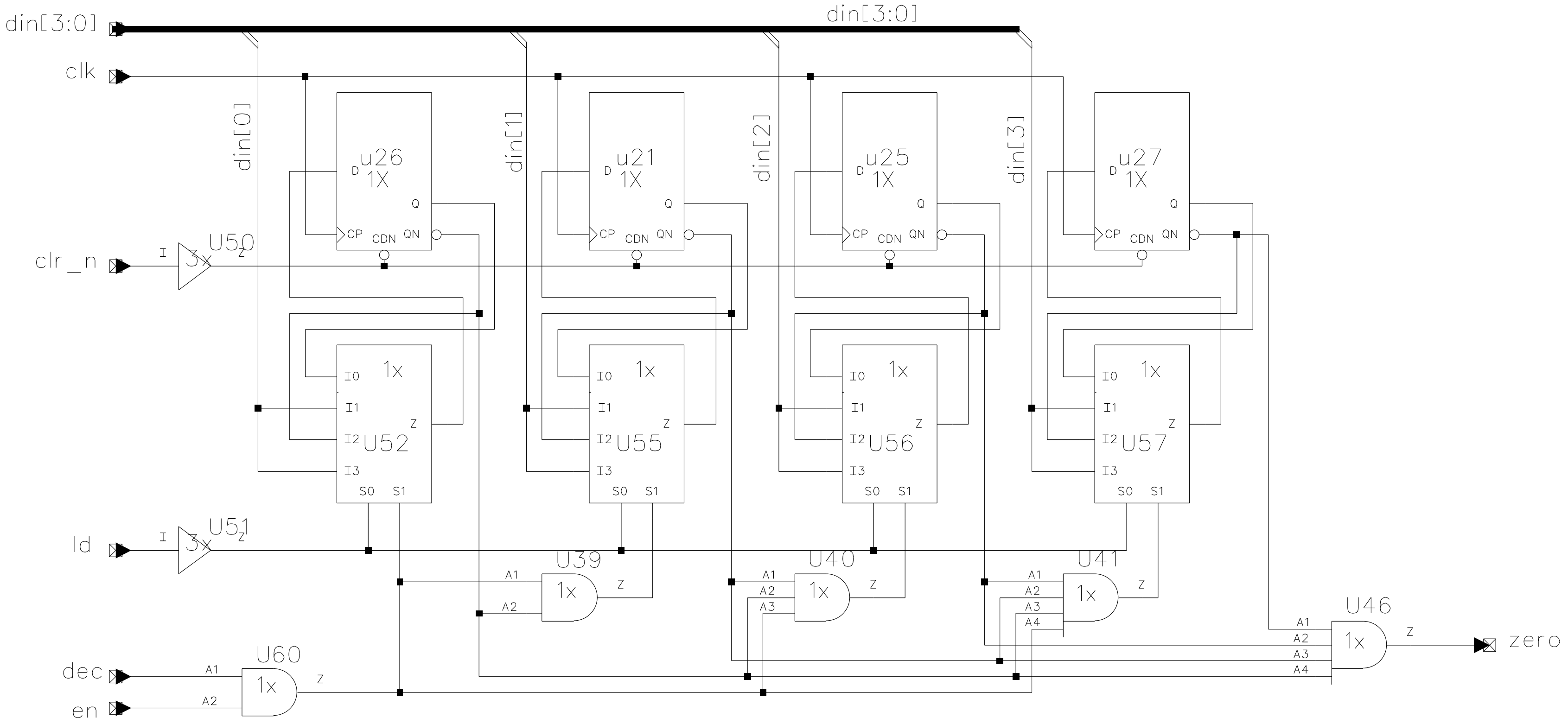
Controller chips

dnctr16_a.ps



Controller chips

[la]dnctr4_a by nhm on 16-Feb-93 at 12:19 A.M.

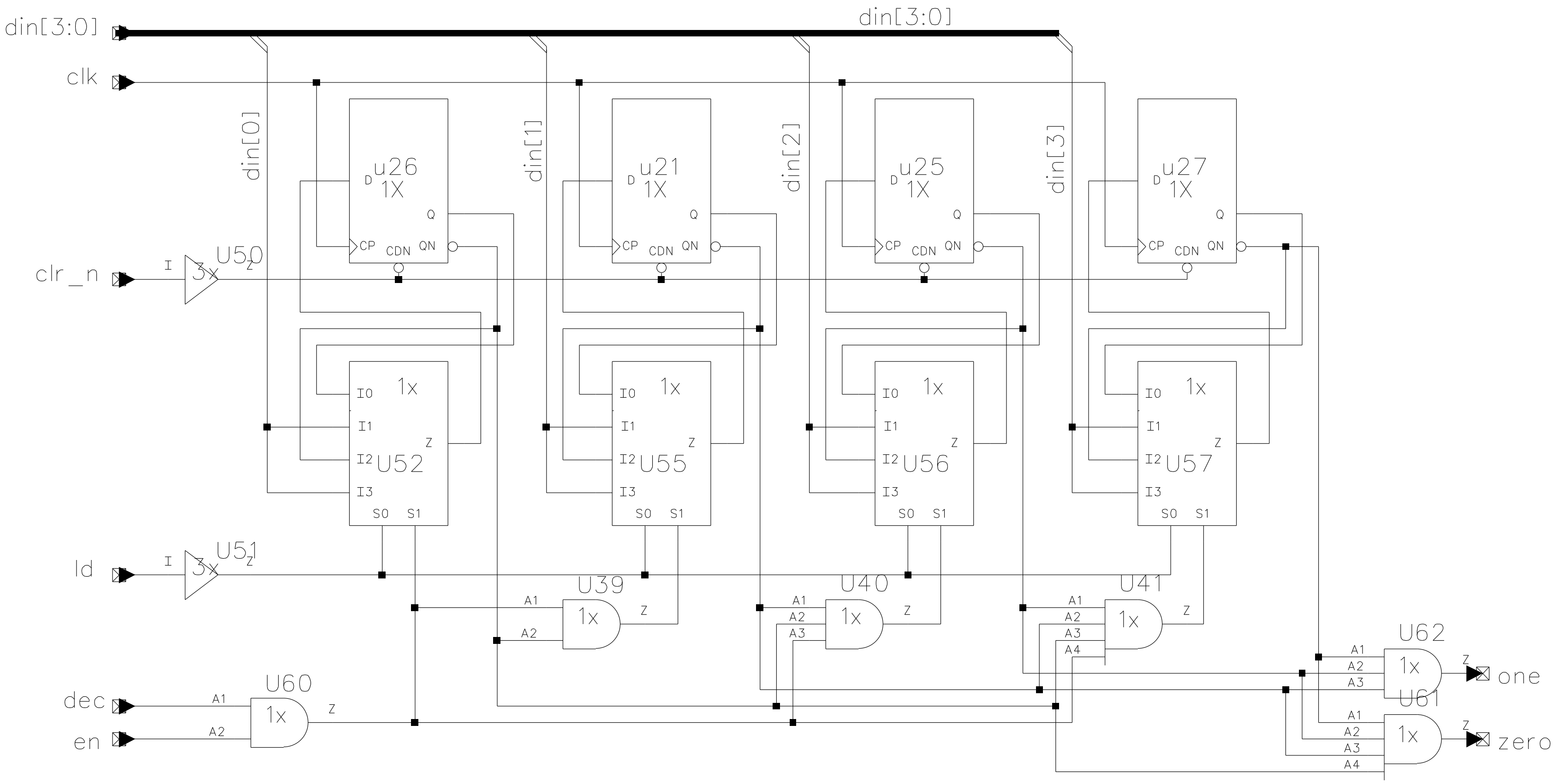


Controller chips

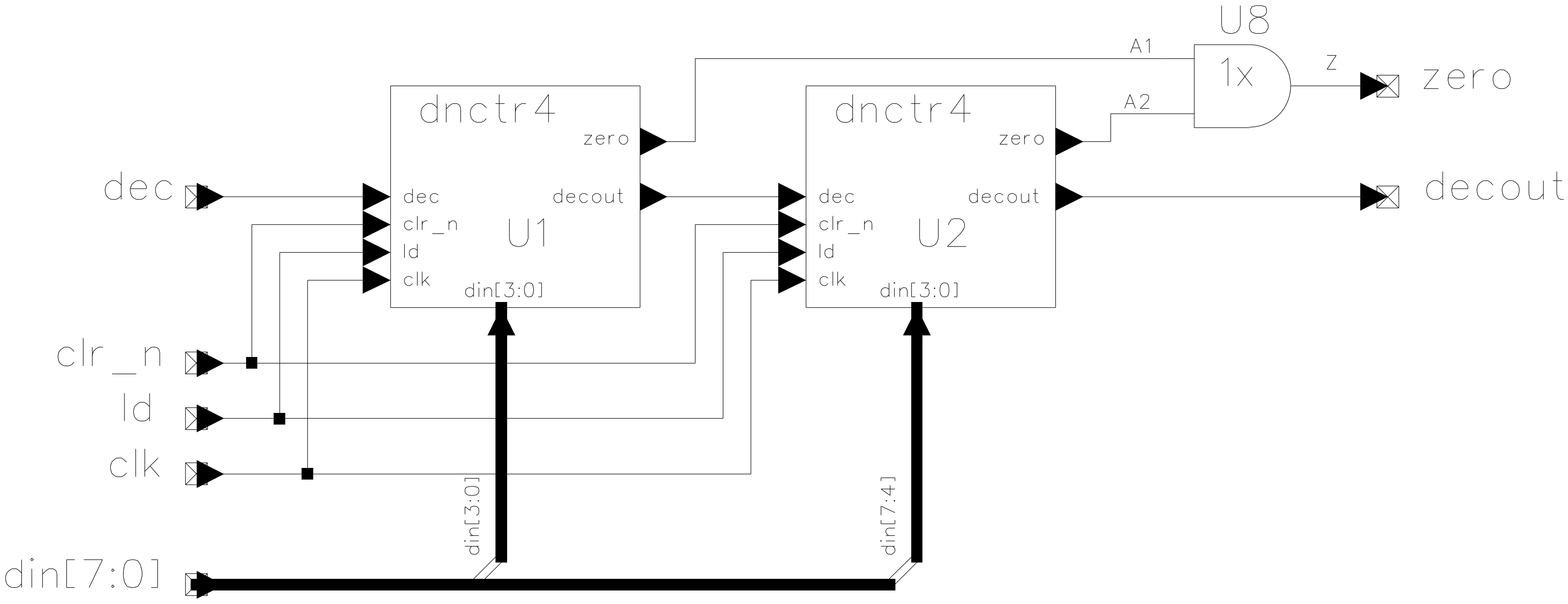
dnctr4_a.ps

[la]dnctr4_b by nhm on 16-Feb-93 at 12:19 A.M.

dnctr4_b.ps



Controller chips

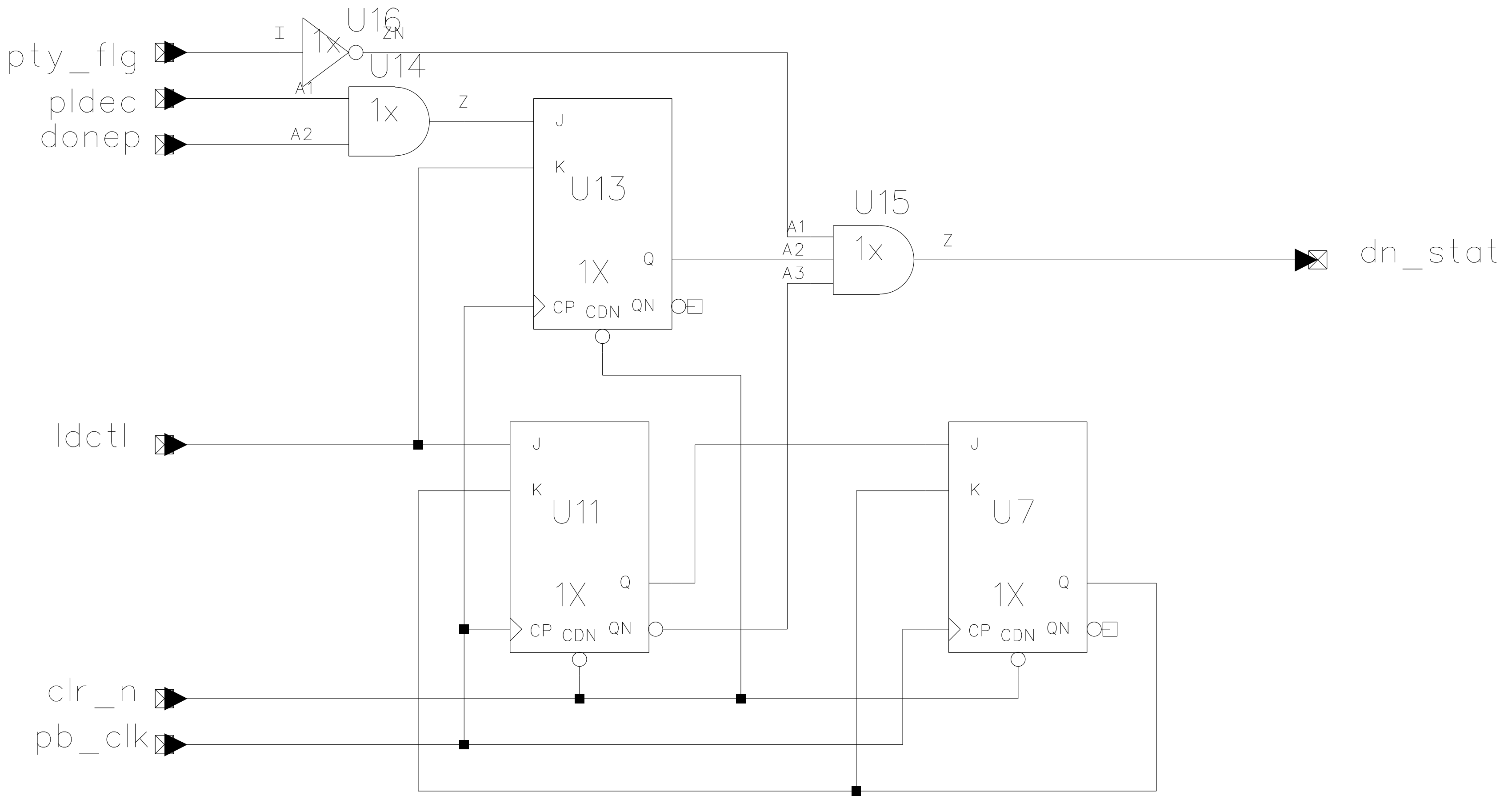


Controller chips

dnctr8.ps

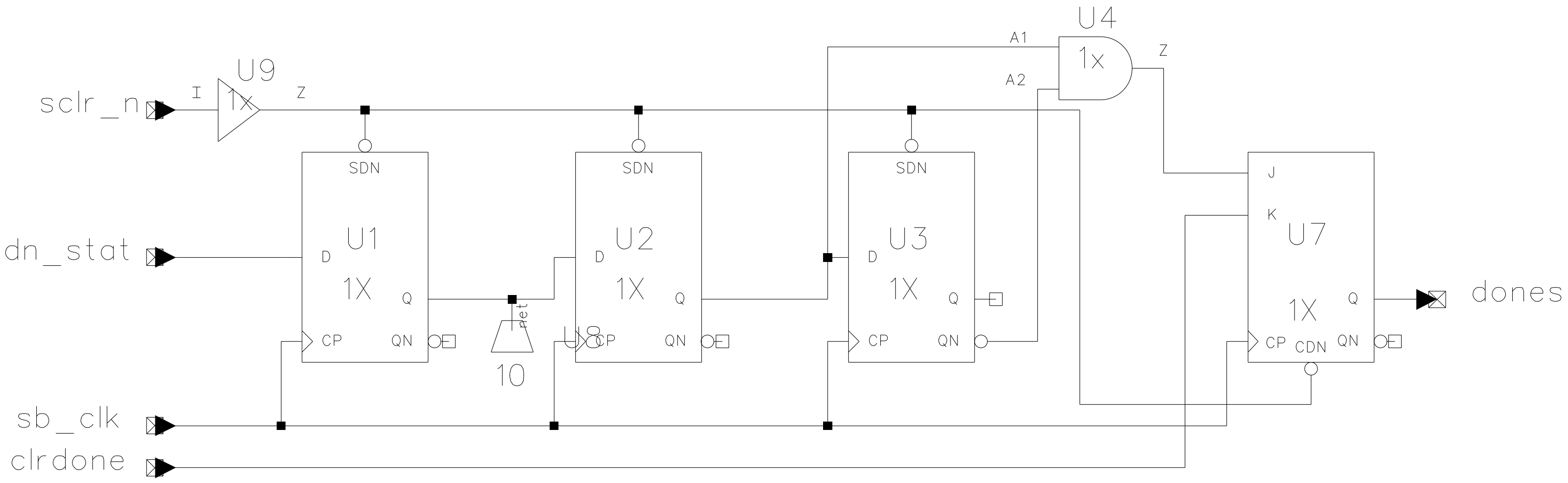
[la]donep_stat by nhm on 16-Feb-93 at 12:19 A.M.

donep_stat.ps



Controller chips

[la]donesync by nhm on 16-Feb-93 at 12:19 A.M.

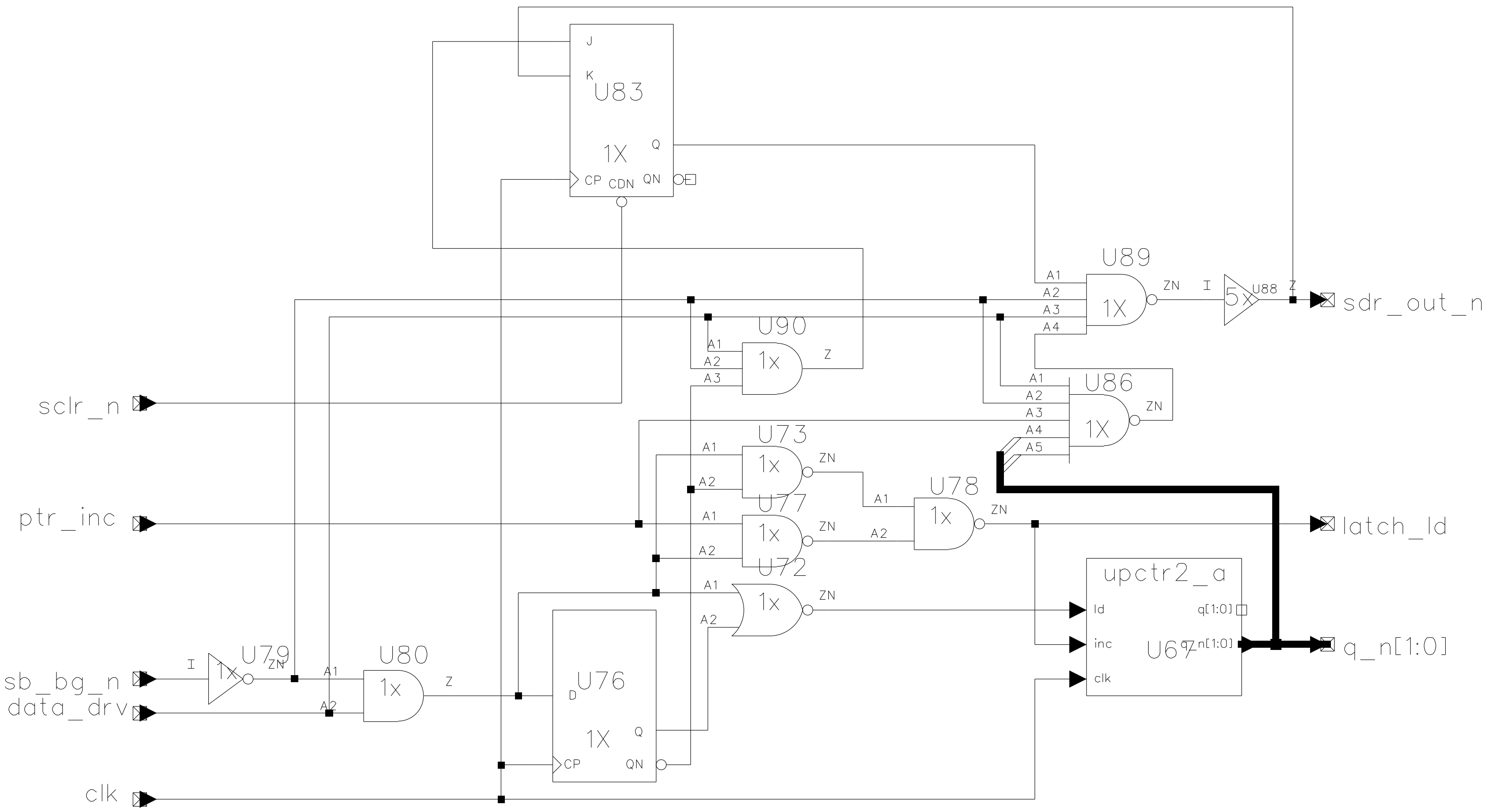


Controller chips

donesync.ps

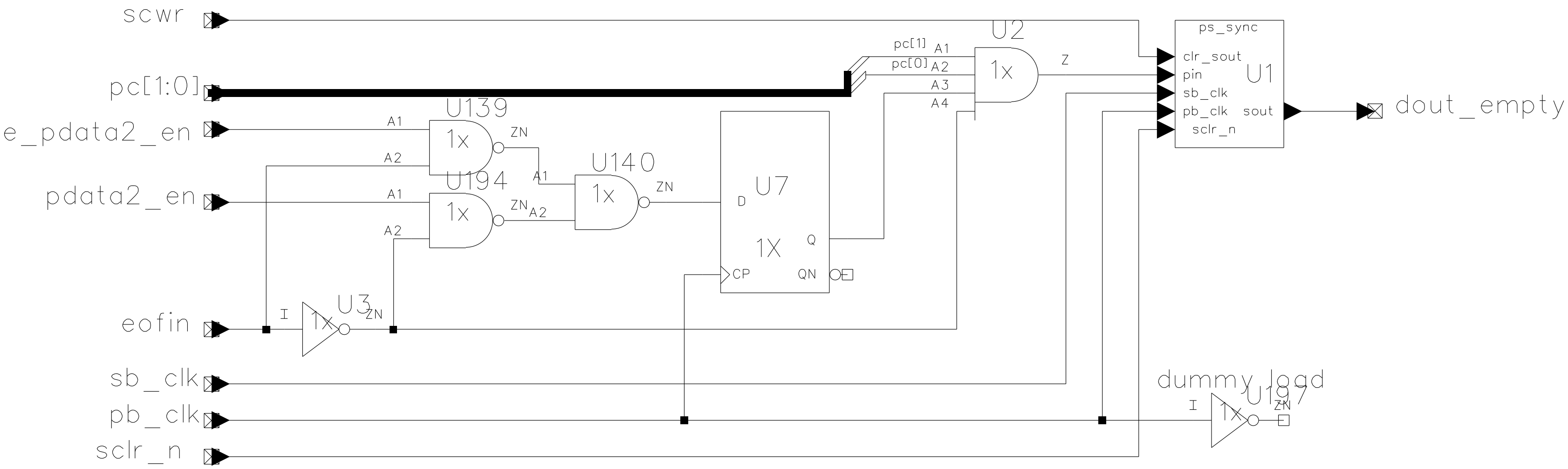
[la]dout_sel by nhm on 16-Feb-93 at 12:19 A.M.

dout_sel.ps



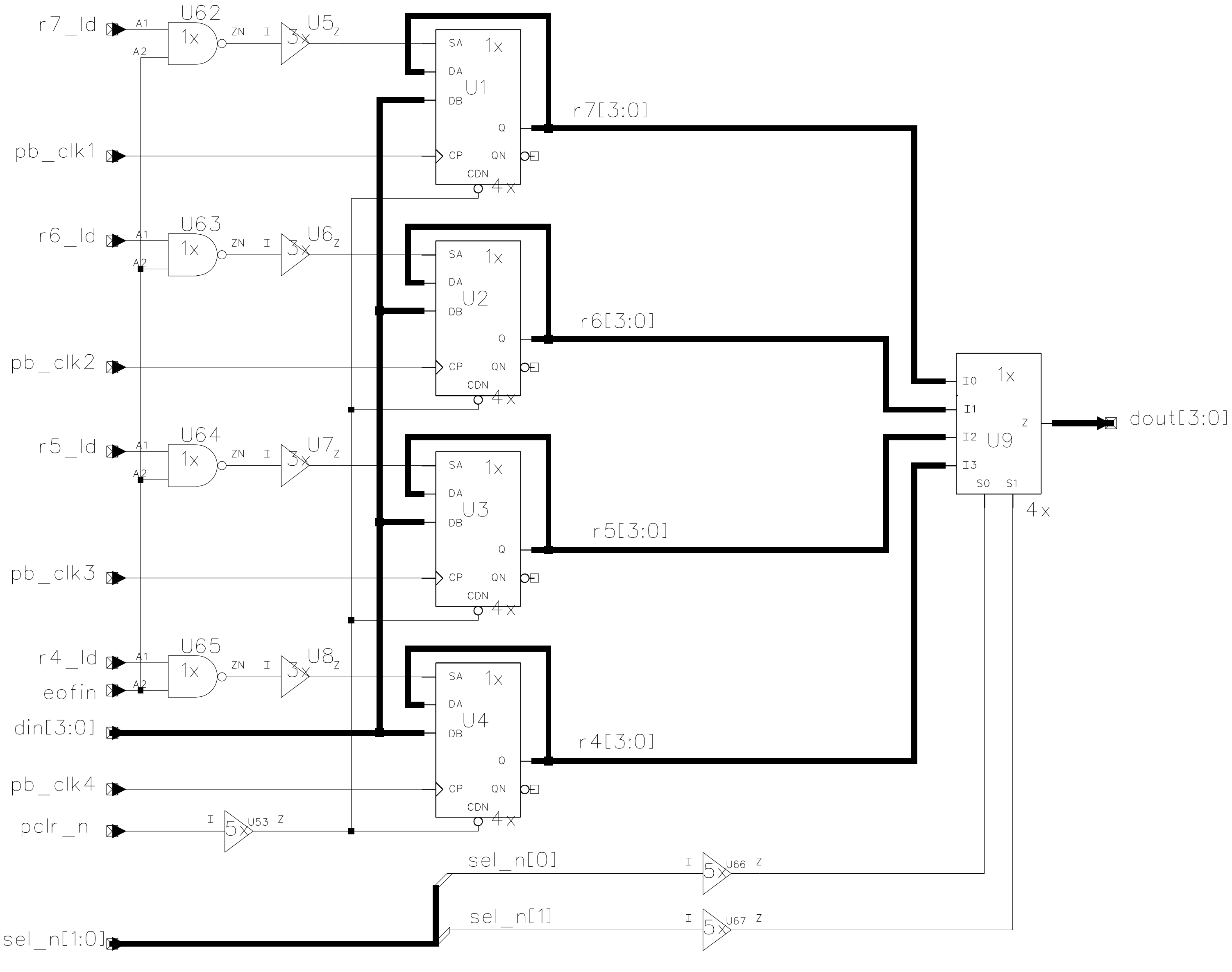
Controller chips

[la]dout_stat by nhm on 16-Feb-93 at 12:20 A.M.



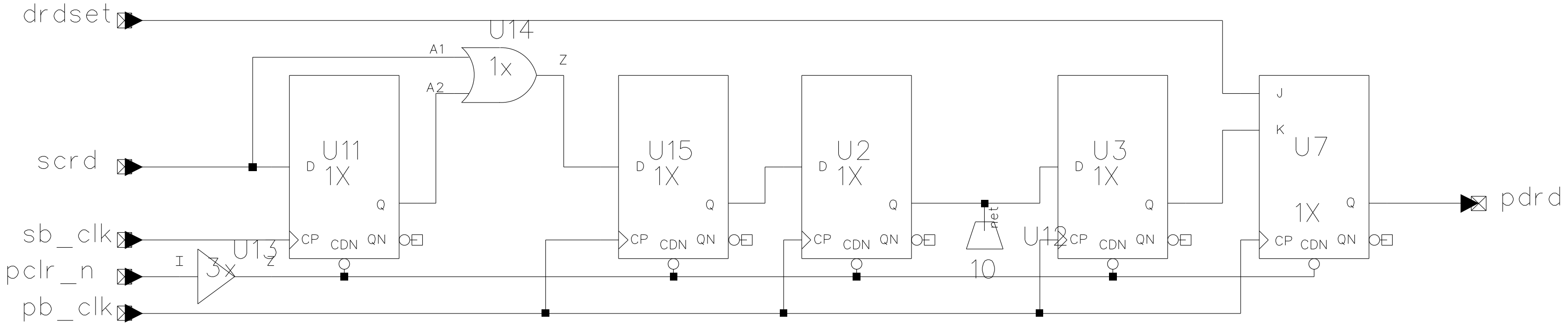
Controller chips

dout_stat.ps



Controller chips

[la]drdsync by nhm on 16-Feb-93 at 12:20 A.M.

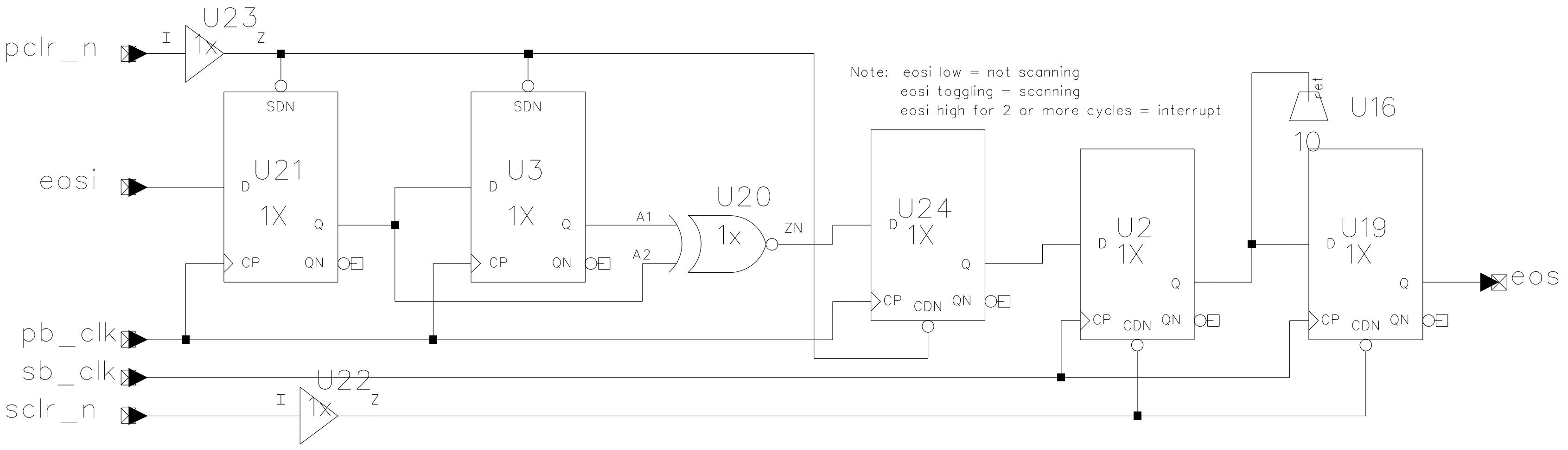


Controller chips

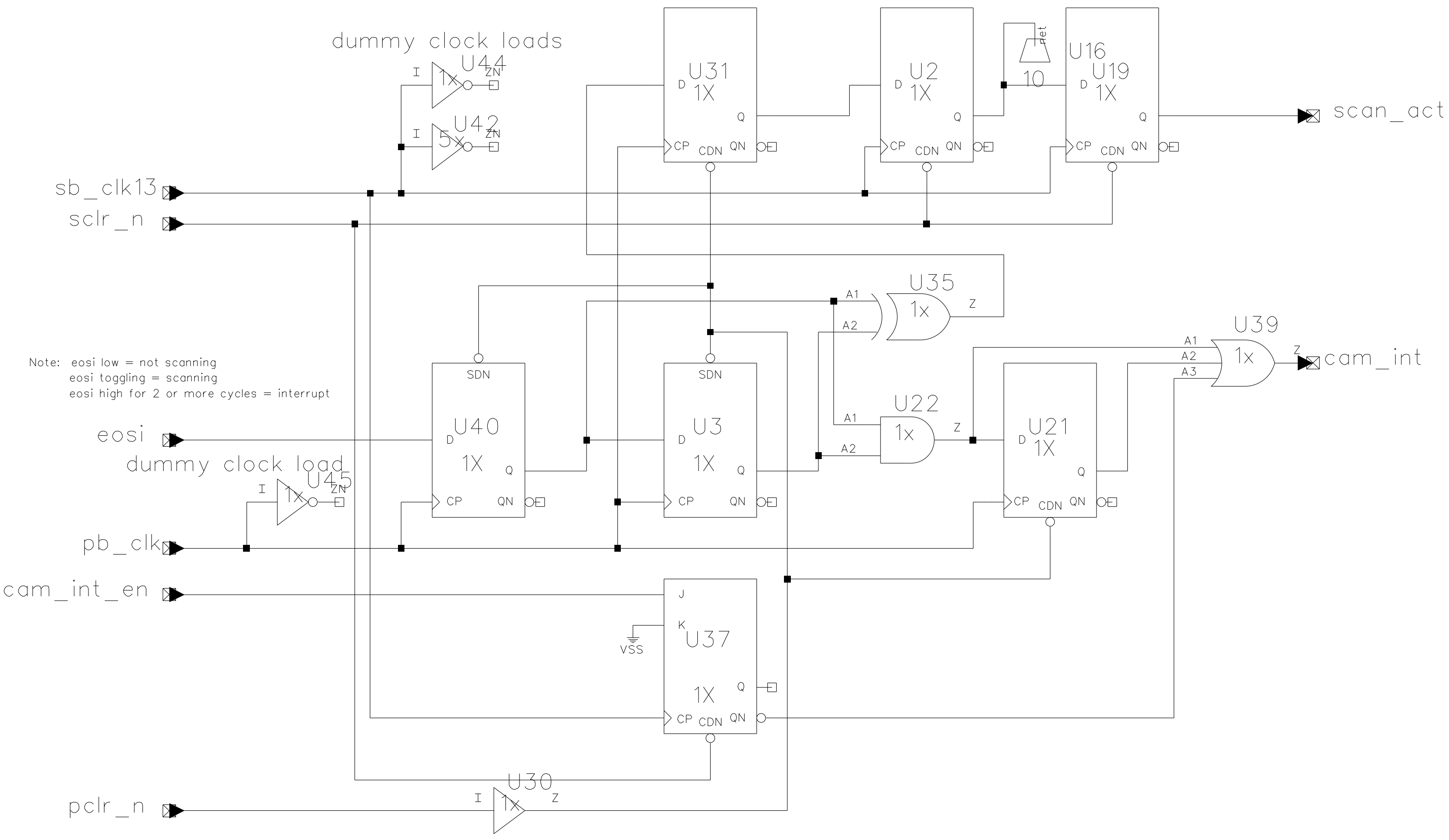
drdsync.ps

[la]eos_dec by nhm on 16-Feb-93 at 12:20 A.M.

eos_dec.ps



Controller chips

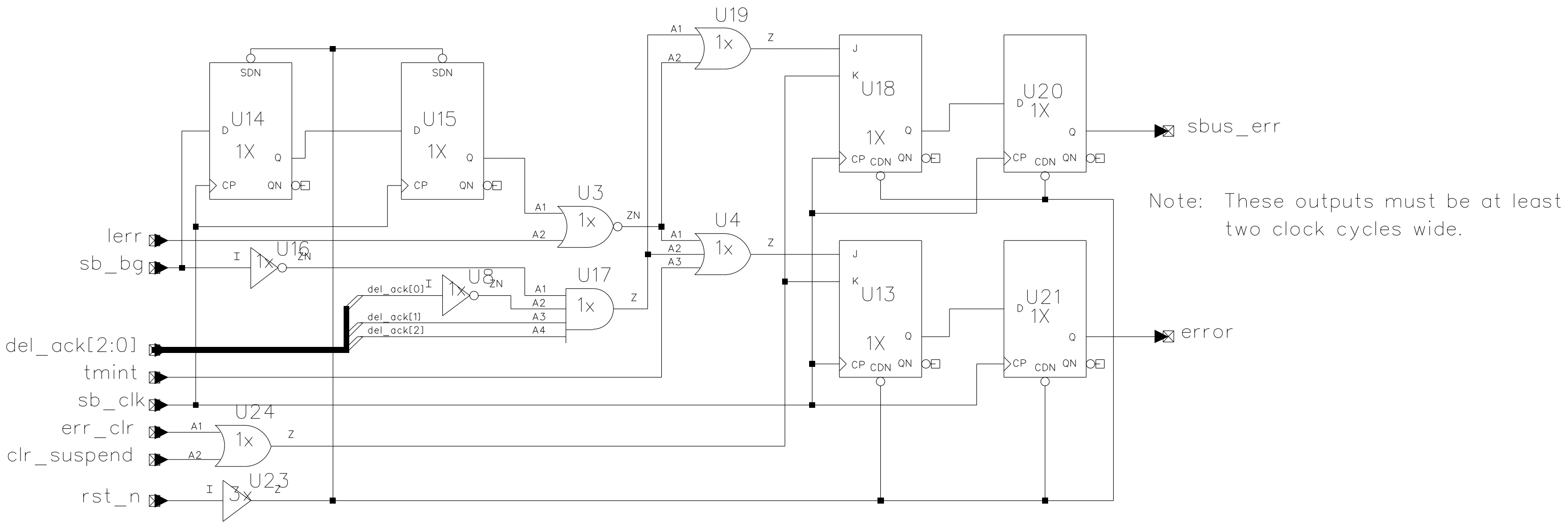


Controller chips

eosi_dcd.ps

[la]errorgen by nhm on 16-Feb-93 at 12:20 A.M.

errorgen.ps



Controller chips

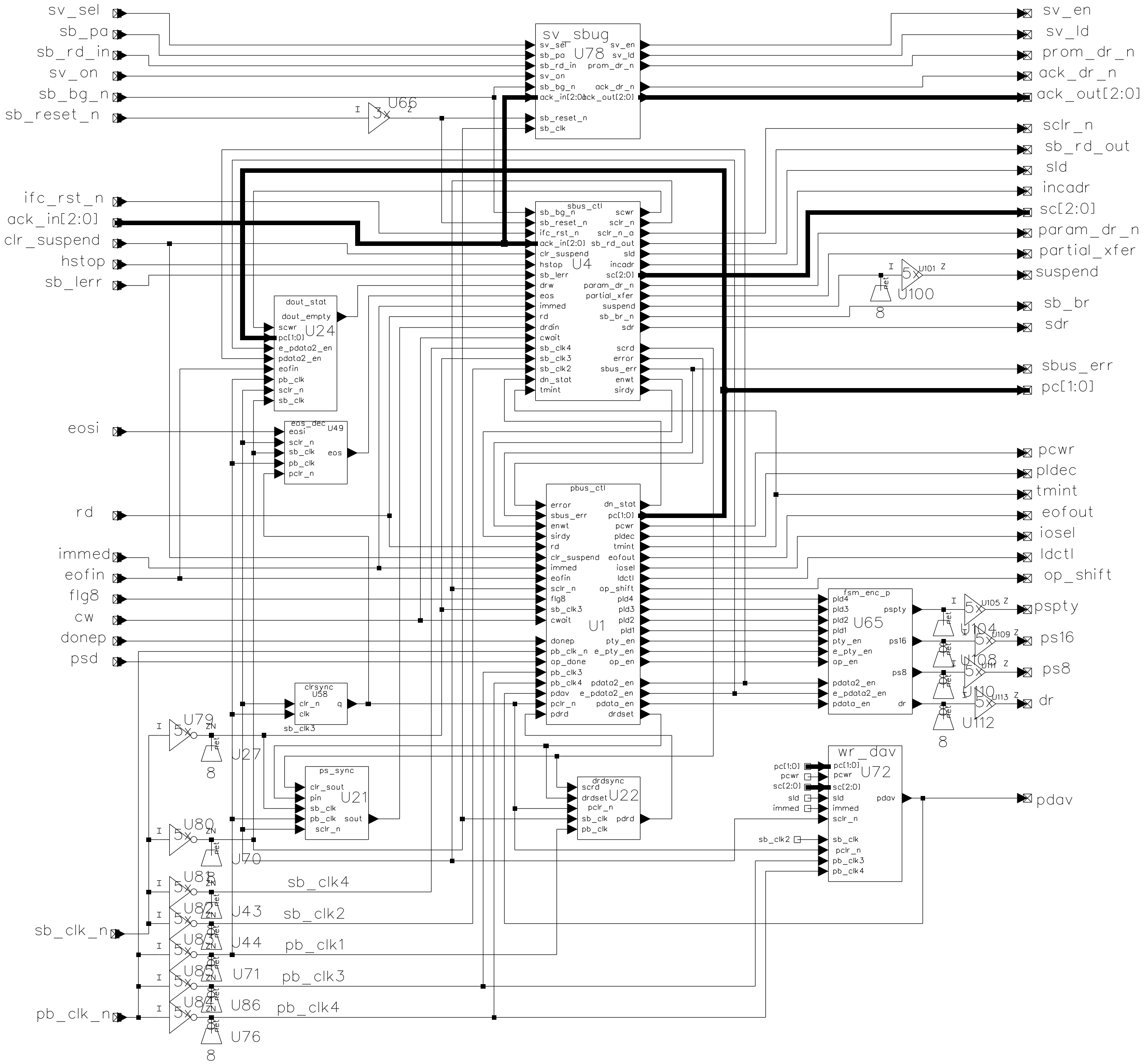
```
#cell2 * fsm_dec sm * 19 any 0 v8r2.2
# "26-Nov-91 GMT" "18:18:12 GMT" "26-Nov-91 GMT" "18:18:12 GMT" faust * .
sm fsm_dec;

# Input file for the decoding of the FSM output signals
# AUTHOR : David Harnanan
# FILENAME : /im/cam8/interface/fsm_dec.sm
# DATE : 19/7/90       REVISED : 11/26/91

# 4/23/91 faust
#  - revised fsm_enc to provide for simultaneous occurrence
#    of pty_en and (pdata_en or pdata2_en).  No change to
#    this fsm necessary.
#  - changed signal name drop to op_en
#  - changed signal name drpty to pty_en
#  - changed signal name pdrdata to pdata_en
#  - changed signal name pdr2 to pdata2_en

# 9/11/91 faust
#  - revised logic to decode e_pty_en (to be ANDed with eofin)
#  -   and e_op_en (to be ANDed with eofin).

# 11/26/91 faust
#  - removed output e_op_en (not needed).

inputs dr ps8 ps16 pspty;

outputs
        op_en_n
        pty_en_n
        e_pty_en
        pdata_en
        pdata2_en_n
        e_pdata2_en
        pld1
        pld2
        pld3
        pld4 ;

# The encoding is done as follows:
# DR = 1  ==> a drive signal
# DR = 0  ==> a load signal
# PS8 = 1  ==> 8-bit mode and
#              <PS16:PSPTY> = address of nybble
# PS8 = 0 ==> not 8-bit mode and
#             PS16 = 1 ==> 16-bit mode and
#                          PSPTY = high or low byte
#             PS16 = 0 ==> not register mode and
#                          PSPTY = 1 ==> parity
#                          PSPTY = 0 ==> opcode
#
# ALL zeroes imply no load or drive
#
#                             TABLE TO BE DECODED
#
#   o p e    p     p e p    p    p    p    | d    p    p    p
#   p t _    d     d _ l    l    l    l    | r    s    s    s
#   _ y p    a     a p d    d    d    d    |      8    1    p
#   e _ t    t     t d 1    2    3    4    |           6    t
```

```
#   n  e y    a     a a                     |                y
#      n _    _     2 t                     |
#        e    e     _ a                     |
#        n    n     e 2                     |
#                   n _                     |
#                     e                     |
#                     n                     |
#                                           |
#   0  0 0    0     0 0  0    0    0    0   |  0    0    0    0
#   0  0 0    0     0 0  1    1    0    0   |  0    0    1    0
#   0  0 0    0     0 0  0    0    1    1   |  0    0    1    1
#   0  0 0    0     0 0  1    0    0    0   |  0    1    0    0
#   0  0 0    0     0 0  0    1    0    0   |  0    1    0    1
#   0  0 0    0     0 0  0    0    1    0   |  0    1    1    0
#   0  0 0    0     0 0  0    0    0    1   |  0    1    1    1
#   1  0 0    0     0 0  0    0    0    0   |  1    0    0    0
#   0  1 0    0     0 0  0    0    0    0   |  1    0    0    1
#   0  0 0    1     0 0  0    0    0    0   |  1    0    1    0
#   0  0 0    1     1 0  0    0    0    0   |  1    0    1    1
#   0  0 0    0     0 0  0    0    0    0   |  1    1    0    0
#   0  0 1    0     0 0  0    0    0    0   |  1    1    0    1
#   0  0 0    1     0 1  0    0    0    0   |  1    1    1    1

# low true outputs
let op_en_n = !(dr & !ps8 & !ps16 & !pspty);
let pty_en_n = !(dr & !ps8 & !ps16 & pspty);
let pdata2_en_n = !(dr & !ps8 & ps16 & pspty);

# high true outputs
let e_pty_en = dr & ps8 & !ps16 & pspty;
let pdata_en = (dr & !ps8 & ps16) | (dr & ps8 & ps16 & pspty) ;
let e_pdata2_en = dr & ps8 & ps16 & pspty;
let pld1 = (!dr & !ps8 & ps16 & !pspty) | (!dr & ps8 & !ps16 & !pspty);
let pld2 = (!dr & !ps8 & ps16 & !pspty) | (!dr & ps8 & !ps16 & pspty);
let pld3 = (!dr & !ps8 & ps16 & pspty) | (!dr & ps8 & ps16 & !pspty);
let pld4 = (!dr & !ps8 & ps16 & pspty) | (!dr & ps8 & ps16 & pspty);

end
```

```
#cell2 * fsm_enc sm * 35 any 0 v8r2.2
# "15-Jan-92 GMT" "19:19:03 GMT" "15-Jan-92 GMT" "19:19:03 GMT" faust * .
sm fsm_enc;

# Input file for the encoding of the FSM output signals
# AUTHOR : David Harnanan
# FILENAME : /im/cam8/interface/fsm_enc.sm
# DATE : 19/7/90       REVISED : 1/15/92

# 2/28/91 faust
#  - changed signal name drop to op_en
#  - changed signal name drpty to pty_en
#  - changed signal name pdrdata to pdata_en
#  - changed signal name pdr2 to pdata2_en

# 4/23/91 faust
#  - revised logic to provide for simultaneous occurrence
#    of pty_en and (pdata_en or pdata2_en).

# 9/10/91 faust
#  - revised logic to encode e_pty_en (to be ANDed with eofin)
#  -   and e_op_en (to be ANDed with eofin).

# 11/26/91 faust
#  - removed input e_op_en (not needed).

# 1/15/92 faust
#  - added maxdelay to speed things up.

inputs
        op_en
        pty_en
        e_pty_en
        pdata_en
        pdata2_en
        e_pdata2_en
        pld1
        pld2
        pld3
        pld4 ;

outputs dr ps8 ps16 pspty;

maxdelay 1 op_en pty_en e_pty_en pdata_en pdata2_en e_pdata2_en pld1 pld2 pld3 pld4 --> dr ps8 ps16 pspty;

# The encoding is done as follows:
# DR = 1  ==> a drive signal
# DR = 0  ==> a load signal
# PS8 = 1  ==> 8-bit mode and
#               <PS16:PSPTY> = address of nybble
# PS8 = 0 ==> not 8-bit mode and
#              PS16 = 1 ==> 16-bit mode and
#                           PSPTY = high or low byte
#              PS16 = 0 ==> not register mode and
#                           PSPTY = 1 ==> parity
#                           PSPTY = 0 ==> opcode
#
# ALL zeroes imply no load or drive
#
```

```
#                             TABLE TO BE ENCODED
#
#   o  p e    p     p e  p    p    p    p    | d    p    p    p
#   p  t _    d     d _  l    l    l    l    | r    s    s    s
#   _  y p    a     a p  d    d    d    d    |      8    1    p
#   e  _ t    t     t d  1    2    3    4    |           6    t
#   n  e y    a     a a                      |                y
#      n _    _     2 t                      |
#        e    e     _ a                      |
#        n    n     e 2                      |
#                   n _                      |
#                     e                      |
#                     n                      |
#                                            |
#   0  0 0    0     0 0  0    0    0    0    | 0    0    0    0
#   0  0 0    0     0 0  1    1    0    0    | 0    0    1    0
#   0  0 0    0     0 0  0    0    1    1    | 0    0    1    1
#   0  0 0    0     0 0  1    0    0    0    | 0    1    0    0
#   0  0 0    0     0 0  0    1    0    0    | 0    1    0    1
#   0  0 0    0     0 0  0    0    1    0    | 0    1    1    0
#   0  0 0    0     0 0  0    0    0    1    | 0    1    1    1
#   1  0 0    0     0 0  0    0    0    0    | 1    0    0    0
#   0  1 0    x     x x  0    0    0    0    | 1    0    0    1
#   0  0 0    1     0 0  0    0    0    0    | 1    0    1    0
#   0  0 0    1     1 0  0    0    0    0    | 1    0    1    1
#   0  0 0    0     0 0  0    0    0    0    | 1    1    0    0
#   0  0 1    x     x x  0    0    0    0    | 1    1    0    1
#   0  0 0    1     0 1  0    0    0    0    | 1    1    1    1

let dr = op_en | pty_en | e_pty_en |
        pdata_en | e_pdata2_en;
let ps8 = (pld1 != pld2) | (pld3 != pld4) |
        e_pty_en | e_pdata2_en;
let ps16 = !pty_en & (pdata_en | (pld1 & pld2) | pld3 | pld4) |
        e_pdata2_en;
let pspty = pdata2_en | pty_en | pld4 | (pld2 & !pld1) |
        e_pty_en | e_pdata2_en;

end
```

[la]fsmchip by nhm on 16-Feb-93 at 12:20 A.M.

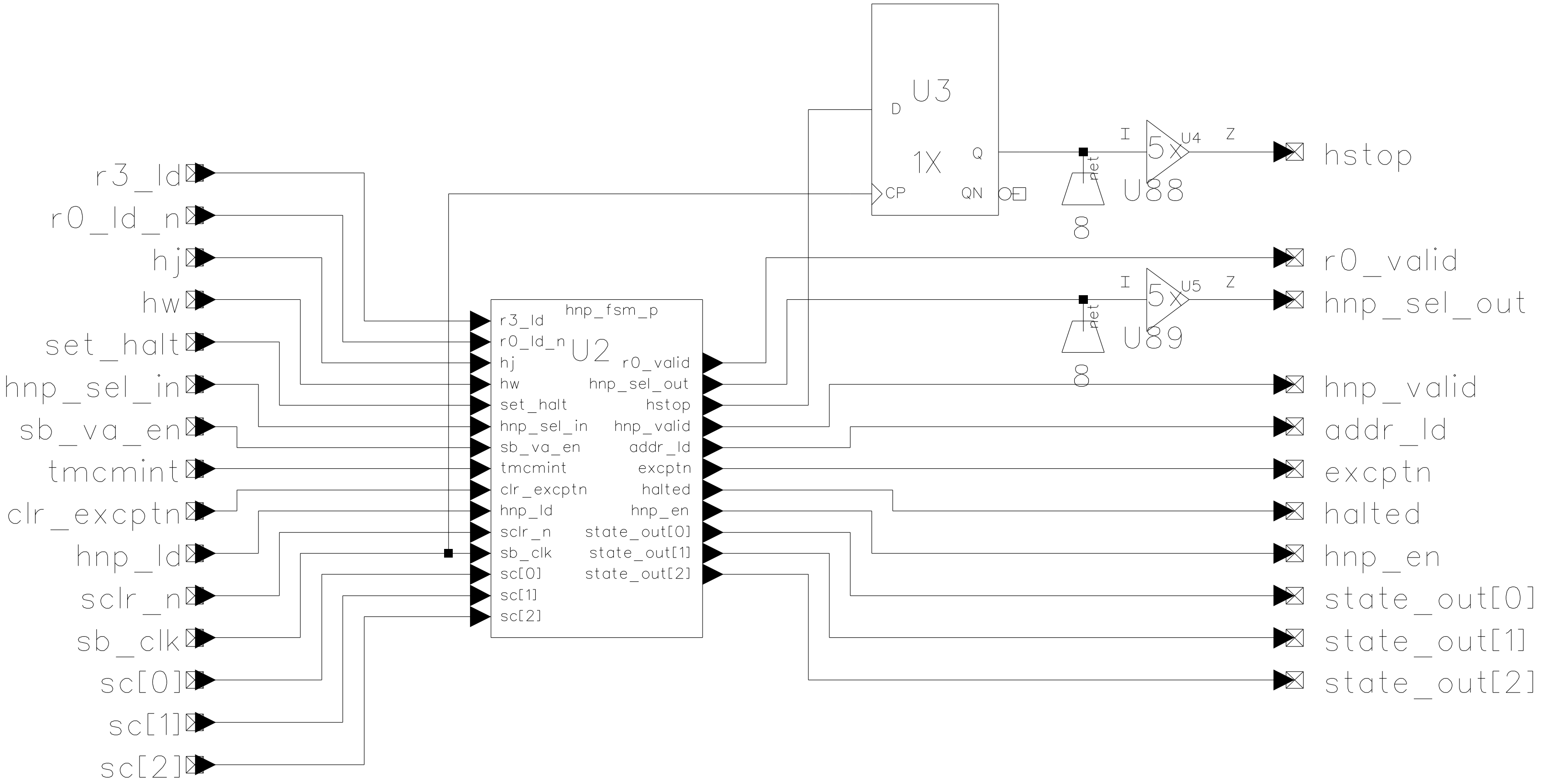


Controller chips

fsmchip.ps

[la]hnp_ctl by nhm on 16-Feb-93 at 12:20 A.M.

hnp_ctl.ps



Controller chips

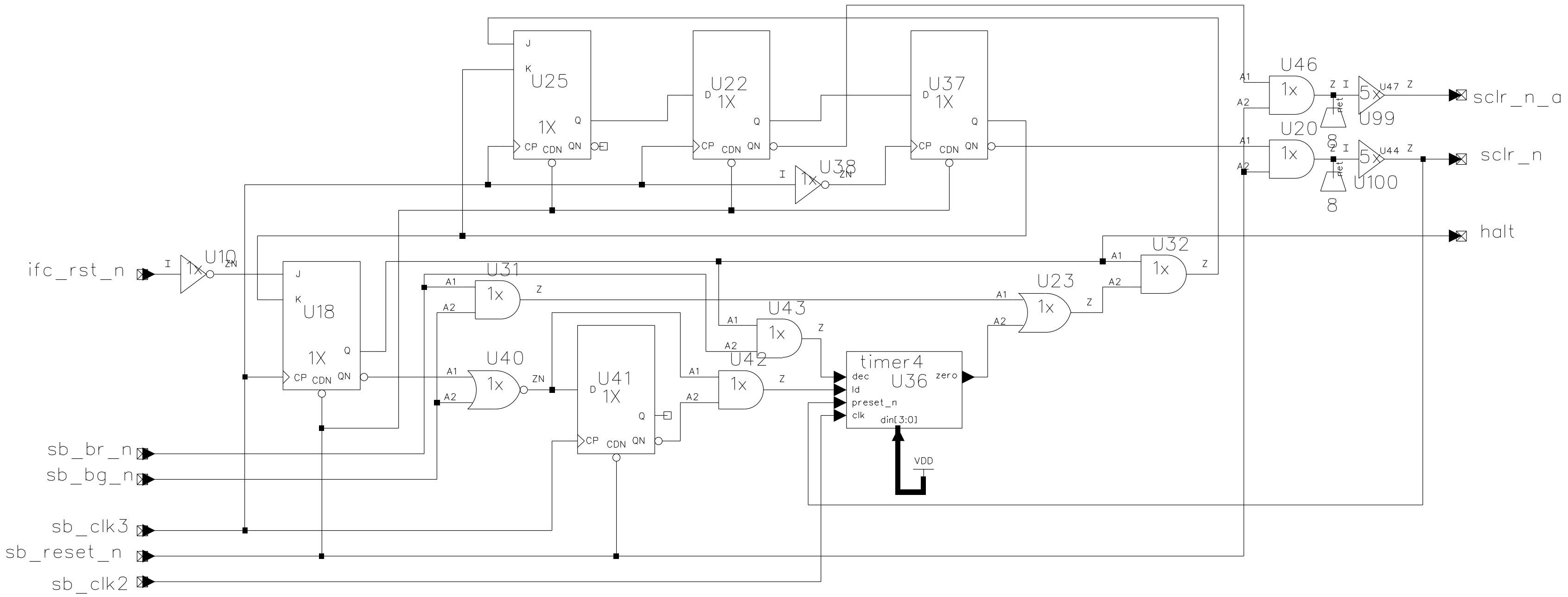
```
#cell2 * hnp_fsm sm * 45 any 0 v8r2.2
# "9-Jan-92 GMT" "18:51:31 GMT" "9-Jan-92 GMT" "18:51:31 GMT" faust * .
sm hnp_fsm;

# Finite State Machine file for the control of
#   Host Next Pointer (HNP) and halt status
# AUTHOR : Doug Faust
# FILENAME : /im/cam8/interface/hnp_fsm.sm
# DATE : 26 Feb 1991       REVISED : 1/9/92

clock sb_clk 25;

reset !sclr_n --> exception_state;

inputs
        hnp_ld          # load hnp
        clr_excptn      # clear exception status
        tmcmint         # cam timeout or interrupt
        sb_va_en        # sbus virtual address enable
        sc[2:0]         # control reg address counter
        hnp_sel_in      # condition requiring hnp to be driven
        set_halt        # programmed halt pulse
        hw              # host wait bit of control word 0
        hj;             # host jump bit of control word 0

latched inputs
        r0_ld_n         # control register 0 load signal
        r3_ld;          # control register 3 load signal

outputs
        state_out[2:0]  # state outputs
        hnp_en          # enable hnp to sbus data lines
        halted          # interface has gotten an exception,
                        #   cam timeout, cam interrupt, or
                        #   programmed halt.
        excptn          # exception status
        addr_ld         # loads the current and previous
                        #   slist pointers (from sbus data
                        #   when virtual address is driven)
        hnp_valid       # host next pointer has been written
        hstop           # interface goes to wait state after
                        #   the current instruction is done.
        hnp_sel_out;    #  condition requiring hnp to be driven
                        #    (output from the low-order chip)
rs=0 outputs
        r0_valid;       # internal status to tell when to use hw and hj bits

let addr_ld = sb_va_en & sc[0] & sc[1];

state exception_state
        {
        let state_out[2] = 0;
        let state_out[1] = 0;
        let state_out[0] = 0;
        let excptn = 1;
        let halted = 1;
        let hstop = 1;
        let hnp_sel_out = r0_valid & (hw | hj);
        }
```

```
        # allow use of hnp if sctlfsm still running
        addr_ld & !sc[2] & hnp_sel_in --> exception_state hnp_en r0_valid=0,
        clr_excptn --> halt_state,
                --> exception_state;

state halt_state
        {
        let state_out[2] = 0;
        let state_out[1] = 1;
        let state_out[0] = 0;
        let halted = 1;
        let hstop = 1;
        let hnp_sel_out = 1;
        }

        tmcmint --> exception_state,
        hnp_ld --> hnp_start_state,
                --> halt_state;

state hnp_start_state
        {
        let state_out[2] = 1;
        let state_out[1] = 0;
        let state_out[0] = 1;
        let hnp_valid = 1;
        let hnp_sel_out = 1;
        }

        tmcmint --> exception_state,
        set_halt --> halt_state,
        addr_ld & !sc[2] & hnp_sel_in --> hnp_driven_state hnp_en r0_valid=0,
                --> hnp_start_state;

state hnp_driven_state
        {
        let state_out[2] = 1;
        let state_out[1] = 1;
        let state_out[0] = 0;
        let hnp_sel_out = 1;
        let hstop = r0_valid & hw;
        let hnp_valid = 1;
        }

        tmcmint --> exception_state,
        set_halt | (hw & r0_valid) --> halt_state,
        !r0_ld_n --> hnp_driven_state r0_valid=1,
        r3_ld --> hnp_done_state,

        # this line enables the hnp after rerun or error
        #   acks during a control burst
        addr_ld & !sc[2] & hnp_sel_in --> hnp_driven_state hnp_en r0_valid=0,
        hnp_ld --> hnp_valid_state,
                --> hnp_driven_state;

state hnp_done_state
        {
        let state_out[2] = 1;
        let state_out[1] = 1;
```

```
        let state_out[0] = 1;
        let hnp_sel_out = r0_valid & (hw | hj);
        let hstop = r0_valid & hw;
        }

        tmcmint --> exception_state,
        set_halt | (hw & r0_valid) --> halt_state,
        hnp_ld --> hnp_valid_state,
                --> hnp_done_state;

state hnp_valid_state
        {
        let state_out[2] = 1;
        let state_out[1] = 0;
        let state_out[0] = 0;
        let hnp_sel_out = r0_valid & (hw | hj);
        let hnp_valid = 1;
        }

        tmcmint --> exception_state,
        set_halt --> halt_state,
        !r0_ld_n --> hnp_valid_state r0_valid=1,
        addr_ld & !sc[2] & hnp_sel_in --> hnp_driven_state hnp_en r0_valid=0,
                --> hnp_valid_state;
end

# 3/4/91 (faust)
#   - added r0_ld_n input and r0_valid output.  Fixed bug which
#     allowed invalid (leftover from previous instructions) values
#     of the hw and hj bits to be used.

# 4/4/91 (faust)
#   - added r3_ld input.  Added the hnp_done_state, which is
#     entered when the hnp is driven onto the bus and control
#     register 3 has been loaded, indicating successful
#     completion of a control word burst transfer.
#   - eliminated the sbus_err input since this no longer causes
#     an exception status.

# 4/23/91 (faust)
#   - modified exception_state to allow the use of the hnp if
#     the sctlfsm just misses an hstop signal and starts loading
#     control words.

# 5/29/91 (faust)
#   - added state outputs for test purposes.

# 6/5/91 (faust)
#   - fixed bug which caused r0_valid to be cleared whenever
#     the halt state was entered, even if it was entered because
#     of a hw bit being received.  If the hnp was then written,
#     causing the start state to be entered, and a cam interrupt
#     occurred, the exception state would be entered with
#     r0_valid zero.  If the sctlfsm was still running at this
#     point, it wouldn't use the hnp because it thought it was
#     invalid (whew!).  Fixed by clearing r0_valid only when
#     the hnp is driven.

# 1/9/92 (faust)
```

```
#   - Made r0_ld_n and r3_ld latched inputs for timing purposes.
```

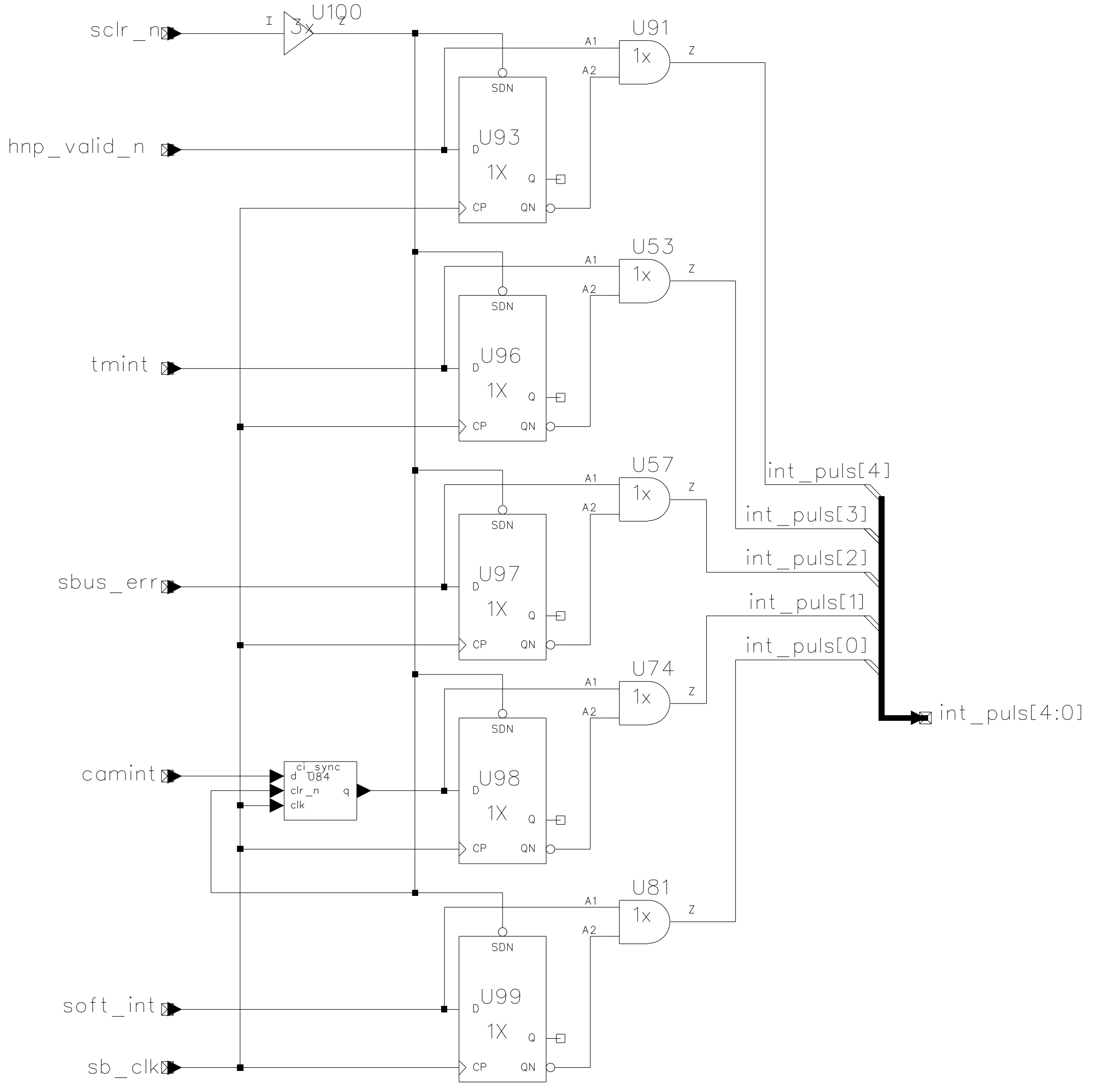
[la]ifc_rst by nhm on 16-Feb-93 at 12:20 A.M.



Controller chips

ifc_rst.ps

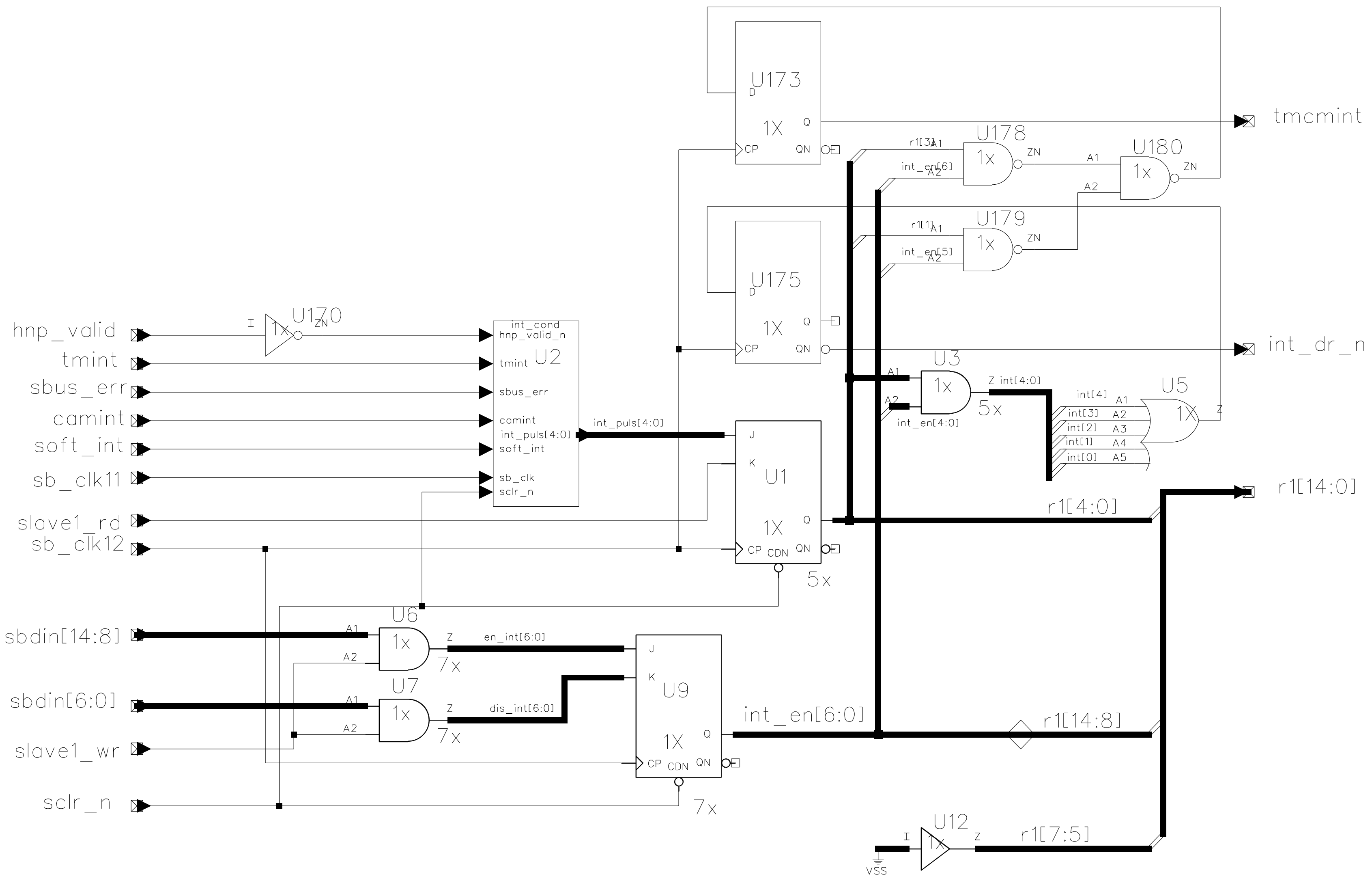
[la]int_cond by nhm on 16-Feb-93 at 12:20 A.M.



int_cond.ps

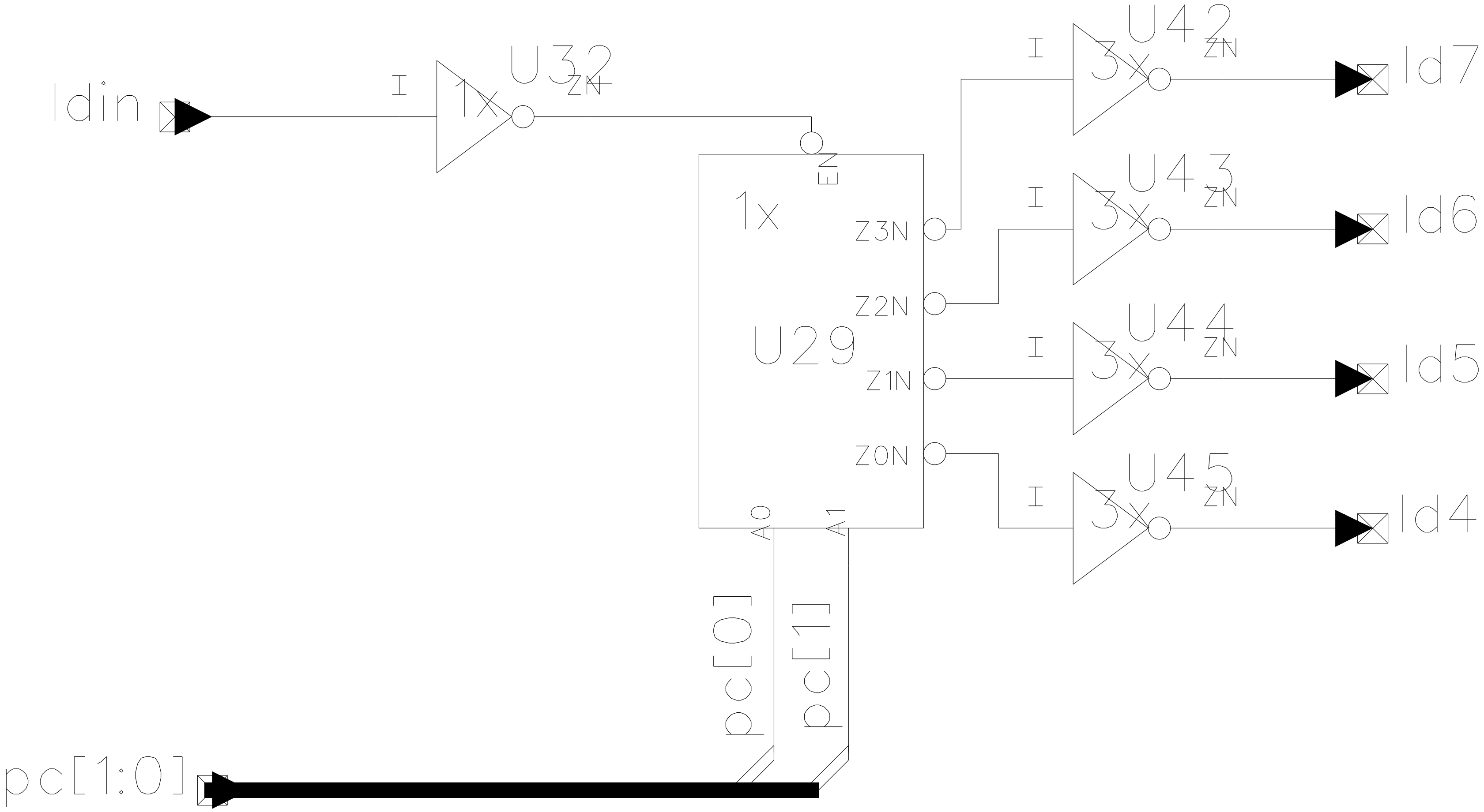
Controller chips

[la]int_reg by nhm on 16-Feb-93 at 12:20 A.M.



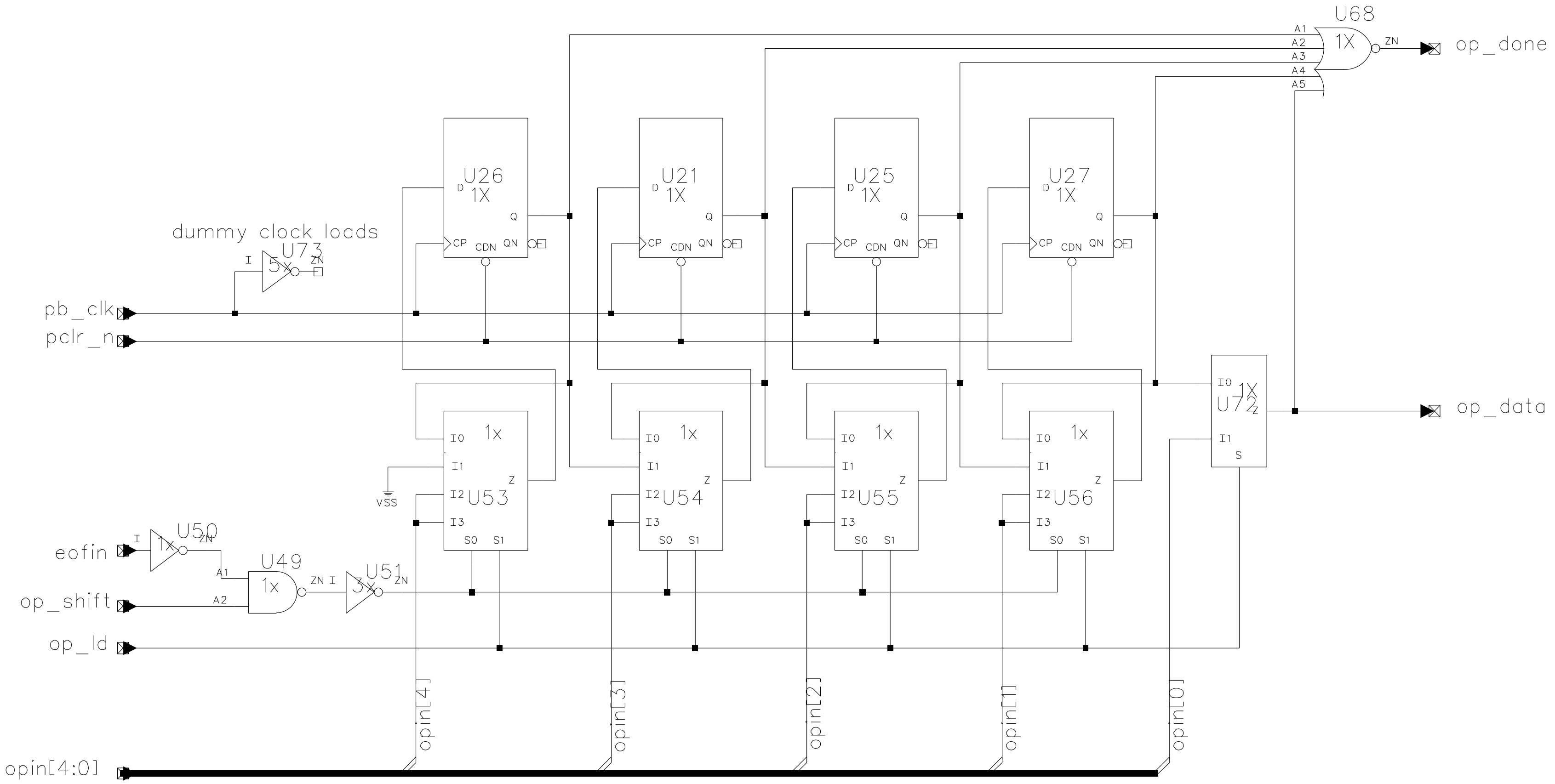
Controller chips

int_reg.ps



Controller chips

[la]opcode_reg by nhm on 16-Feb-93 at 12:20 A.M.

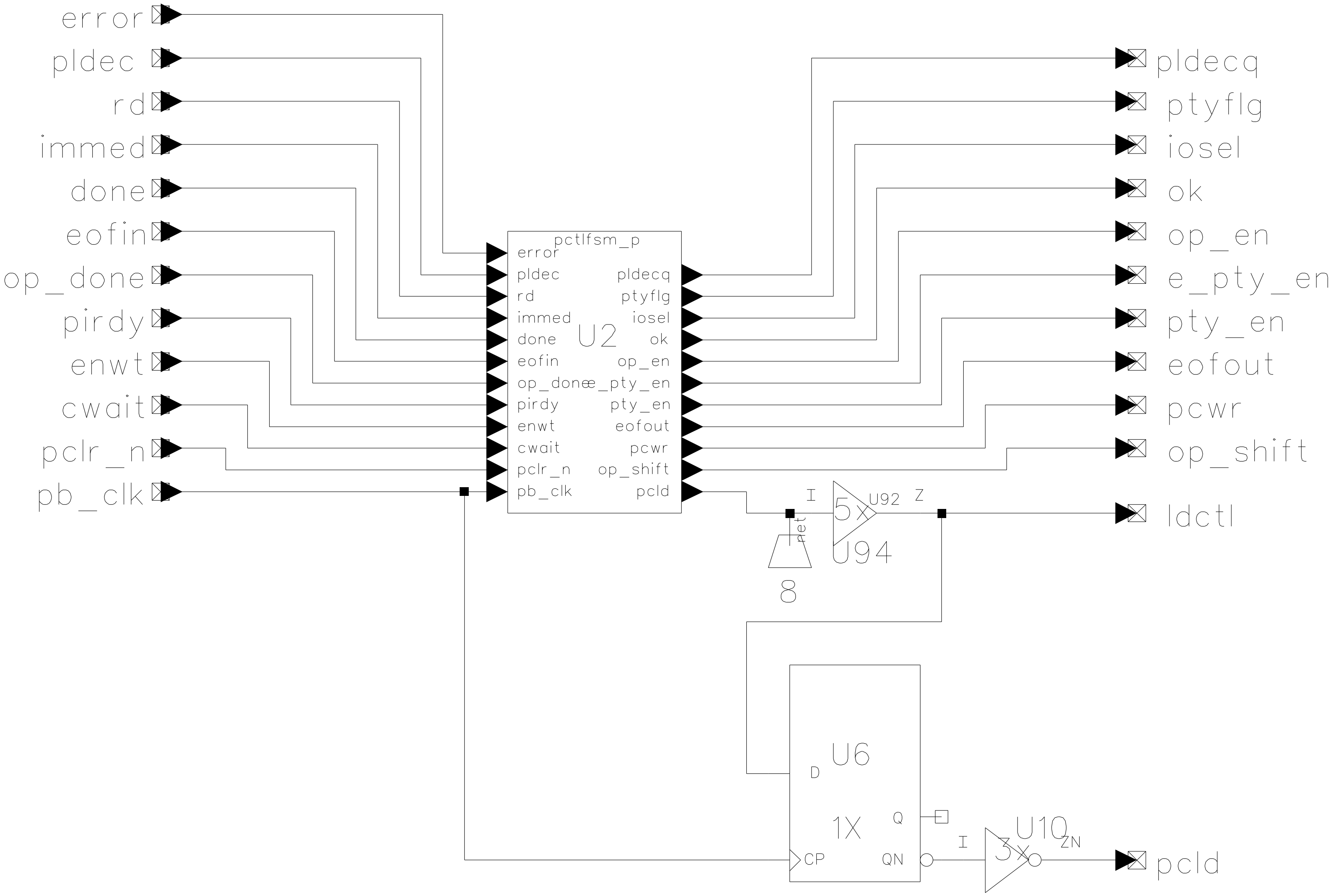


Controller chips

opcode_reg.ps

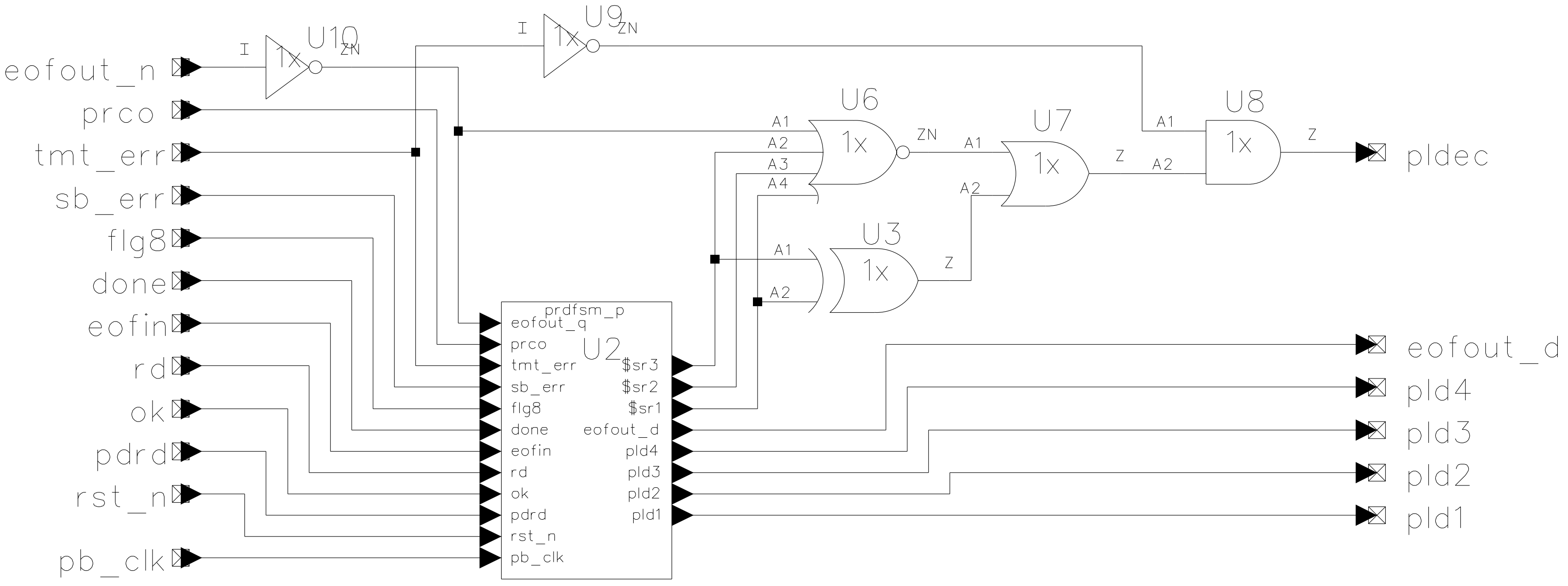
[la]pb_ctl by nhm on 16-Feb-93 at 12:20 A.M.

pb_ctl.ps



Controller chips

[la]pb_rd by nhm on 16-Feb-93 at 12:20 A.M.

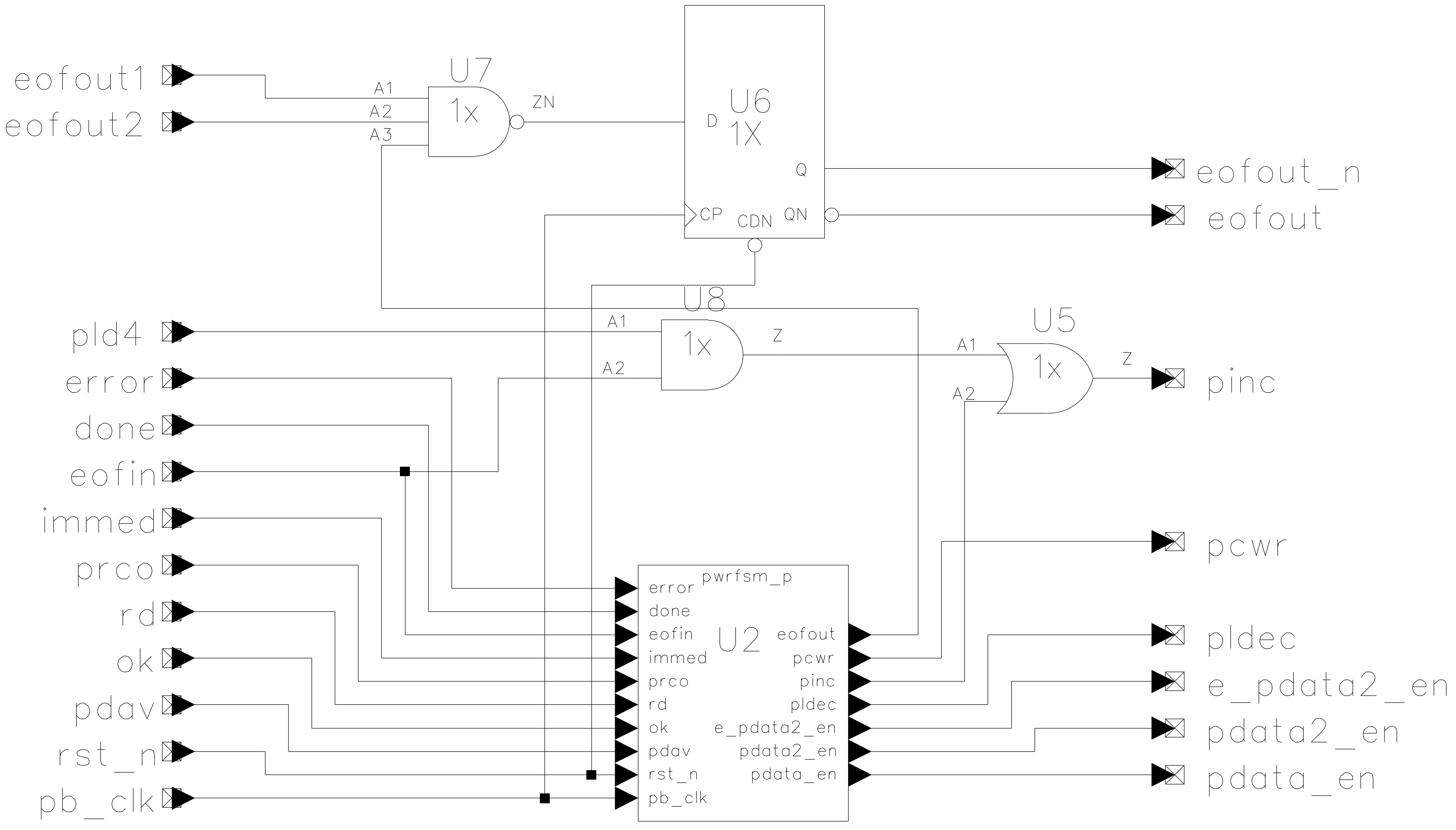


Controller chips

pb_rd.ps

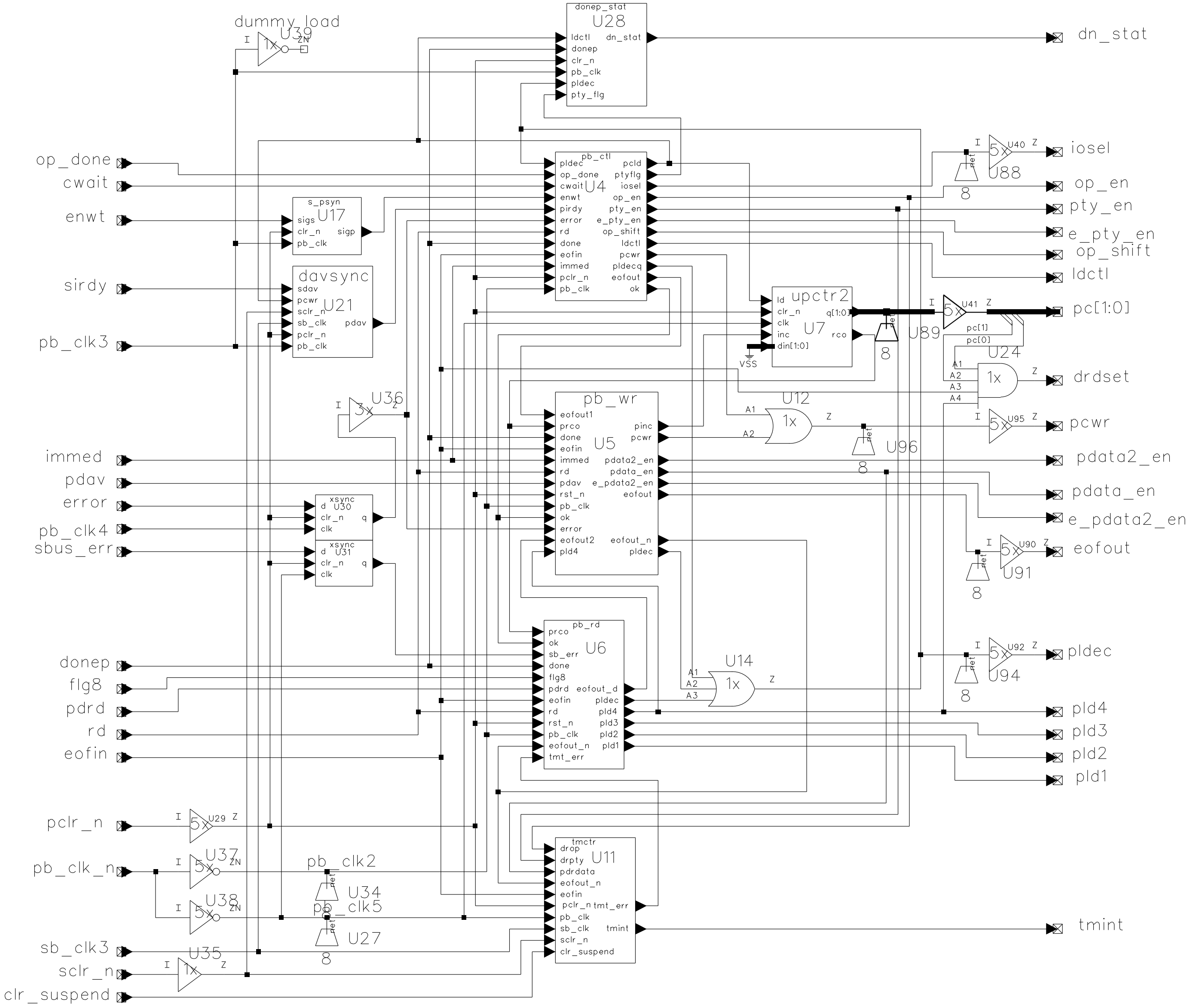
[la]pb_wr by nhm on 16-Feb-93 at 12:21 A.M.

pb_wr.ps



Controller chips

[la]pbus_ctl by nhm on 16-Feb-93 at 12:21 A.M.



Controller chips

pbus_ctl.ps

```
#cell2 * pctlfsm sm * 81 any 0 v8r2.5
# "4-May-92 GMT" "11:31:53 GMT" "4-May-92 GMT" "11:31:53 GMT" faust * .
sm pctlfsm;

# Finite State Machine file for the Control FSM of the PFSM System
# AUTHOR : David Harnanan
# FILENAME : /im/cam8/interface/pctlfsm.sm
# DATE : 14 April 1990       REVISED : 5/4/92

clock pb_clk 25;

reset !pclr_n --> pidle;


inputs
        cwait           # cam wait bit of control word 0
        enwt            # cam wait enable from sctlfsm
        pirdy           # an instruction (4 words) has been loaded
        op_done         # opcode shifting completed
        eofin           # handshake input from pbus
        done            # transfer counter has hit 0
        immed           # immediate bit of control word 0
        rd              # read bit of control word 0
        pldec           # decrement word count
        error;          # Sbus error or cam timeout

outputs
        pcld            # clear pirdy, make done status pulse if
                        #   transfer count is 0, and load opcode shift
                        #   reg and transfer count regs.
        op_shift        # opcode shift enable
        pcwr            # reset pdav status (data available from Sbus)
        eofout=1        # handshake output to pbus
        pty_en          # parity bit enable
        e_pty_en        # parity bit enable
        op_en;          # opcode data enable
##        e_op_en;        # opcode data enable

rs=0 outputs ok iosel ptyflg pldecq;

maxdelay 1 pb_clk --> pcld;
maxdelay 1 eofin --> op_en;

let noop = (rd & immed);

####
#### Note, a davsync should be placed between sirdy and pirdy
#### pcld should clear the pirdy flag of the davsync element
####

state pidle = 001
  {
  let pcwr = !immed & done;
  let e_pty_en = ptyflg;
  let eofout = !(ptyflg & eofin);
  }
  ptyflg & eofin --> pidle,
  ptyflg & !eofin --> pidle ptyflg=0,
  error --> pidle pcld=1 ok=0 iosel=0,
```

```
  pirdy & cwait & enwt --> pidle,
  pirdy & !ptyflg --> load_op pcld=1,
        --> pidle;

# load the opcode into its shift reg
state load_op = 111
  error --> pidle ok=0 iosel=0,
        --> start_op op_en=1 pcld=1;

state start_op = 110
  error --> pidle ok=0 iosel=0,
  noop --> waitdone pldecq=1,
  op_done & rd  --> xmtrd eofout=0,
  op_done & !rd  --> waitdone ok=1 iosel=1 ptyflg=1,
        --> send_op eofout=0 op_en=1 op_shift=1;

state send_op = 010
  {
  let op_shift = !error;
  let op_en = !error;
  }

  error --> pidle pcld=1 ok=0 iosel=0,
  op_done & !rd & !eofin --> waitdone ok=1 iosel=1 ptyflg=1,
  op_done & rd & !eofin --> xmtrd eofout=0,
        --> send_op eofout=0 ;

state xmtrd = 000
  {let op_en = !error;
   let op_shift = 1;}
  error --> pidle pcld=1 ok=0 iosel=0,
  !eofin --> waitdone ok=1 iosel=1 ptyflg=1,
        --> xmtrd eofout=0;

state waitdone = 100
  error --> pidle pcld=1 ok=0 iosel=0 pldecq=0,
  done & !rd & !eofin & pldec --> send_pty,

# this was commented out because it can't happen, 7/2/91
#   done & !rd & !ptyflg --> pidle pcwr=1 ok=0 iosel=0,

  done & noop --> pidle pcwr=1 ok=0 iosel=0 pldecq=0,
  done & rd & eofin & pldec --> rd_done,
        --> waitdone;

state rd_done = 101
  # always wait one extra clock
  !ptyflg --> rd_done ptyflg=1,
      --> send_pty iosel=0 eofout=1;

state send_pty = 011
  --> pidle pcwr=1 ok=0 iosel=0 pty_en=1 eofout=0;
end

# 11/27/90 (faust)
#   - changed wr input to rd
#   - decoded noop internally

# 12/18/90 (faust)
```

```
#   - eliminated ldctl output (was identical to
#     pcld output, so screwed up e2v translator)

# 1/23/91 (faust)
#   - changed pidle state to output pcwr if done is set and no
#     immediate mode.  This resets any pdav which may have
#     occurred when the sctlfsm requested one too many burst
#     transfers.

# 1/31/91 (faust)
#   - changed pidle state to make eofout and drpty independent
#     of eofin.

# 2/12/91 (faust)
#   - added error input (sbus or cam timeout).

# 2/27/91 (faust)
#   - changed error response to clear ptyflg, ok, and
#       iosel outputs.

# 2/28/91 (faust)
#   - changed signal name drop to op_en.
#   - changed signal name drpty to pty_en.
#   - changed signal name pseldone to op_done.
#   - modified to make the op_en and pty_en signals one clock
#     earlier so they can be latched on the dataregs chip.
#   - eliminated the ioout signal which was an unlatched output.
#     this has the effect of adding one clock cycle to writes.

# 3/19/91 (faust)
#   - fixed bug in op_en equations.

# 4/11/91 (faust)
#   - modified so that sending the parity bit is not
#     omitted if there is an error

# 7/2/91 (faust)
#   - removed two lines containing untestable conditions.

# 9/10/91 (faust)
#   - modified to remove eofin as a qualifier of pty_en, op_shift,
#     and op_en; this fixes a timing problem.
#     Added outputs e_pty_en, e_op_en which are gated with
#     eofin in the dataregs chip.

# 10/15/91 (faust)
#   - Added load_op state to allow opcode data to be latched in
#     the dataregs chip for better timing.

# 11/26/91 (faust)
#   - Changed timing of op_done to be one clock earlier.
#     Added input pldec.
#     Changed eofout timing to be one clock early so it can
#     be used on external flipflop.
#     Removed e_op_en output.

# 12/11/91 (faust)
#   - Fixed bug in equation for eofout in pidle state.
```

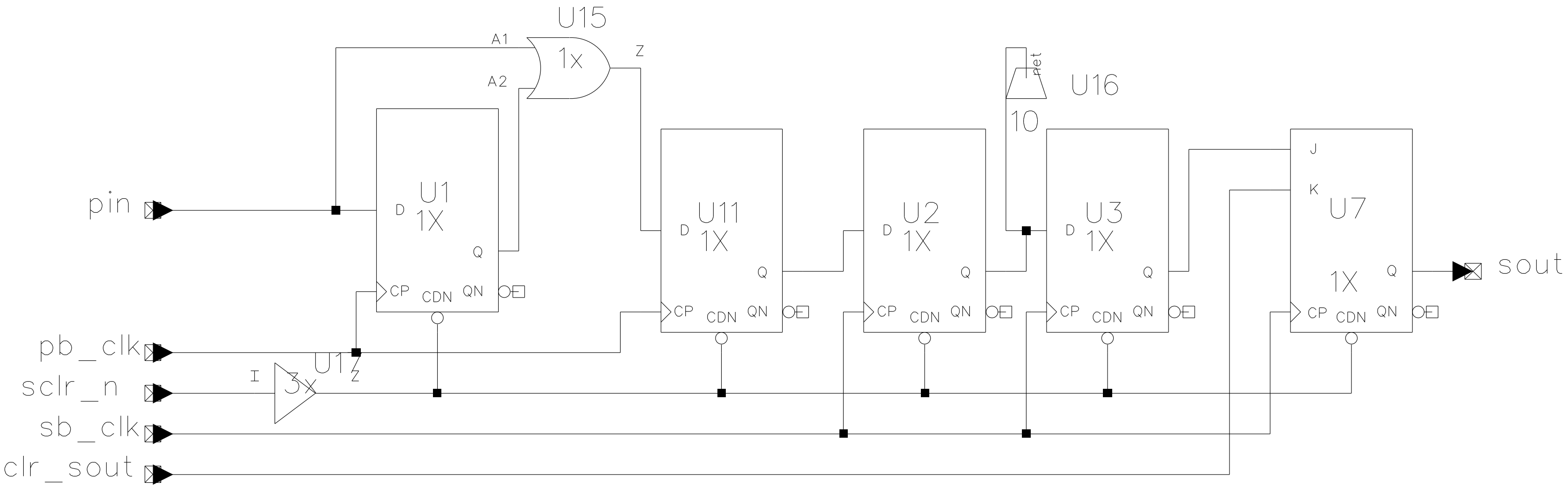
```
# 1/2/92 (faust)
#   - Added output pldecq to decrement counter during noop.

# 1/14/92 (faust)
#   - Changed timing of pcld output to be a clock early, so
#     it can be clocked in ctlsvchip.

# 4/10/92 (faust)
#   - In states load_op and send_op, added !eofin as qualifier
#     to change to next state because of possible cam holdups.
#     Added start_op state to fix a path from eofin to op_done.

# 5/4/92 (faust)
#   - Changed rd_done state to wait an extra clock to allow bus
#     to turn around
```

```
#cell2 * prdfsm sm * 49 any 0 v8r2.2
# "13-Jan-92 GMT" "18:07:20 GMT" "13-Jan-92 GMT" "18:07:20 GMT" faust * .
sm prdfsm;

# Finite State Machine file for the CAM Read FSM of the PFSM System
# AUTHOR : David Harnanan
# FILENAME : /im/cam8/interface/prdfsm.sm
# DATE : 14 April 1990       REVISED : 1/13/92

clock pb_clk 25;

reset !rst_n --> rdwait;

inputs  pdrd            # pbus read buffer full
        ok              # "go" signal from pctlfsm
        rd              # pbus read signal
        eofin           # handshake signal from cam chip
        done            # true when word count==1
        flg8            # 1 if byte transfer mode
        sb_err          # sbus error
        tmt_err         # pbus timeout error
        prco            # carry out from data reg ptr
        eofout_q;       # output of external eofout flipflop

outputs   pld1 pld2 pld3 pld4 eofout_d=1;

state rdwait = 0010
      tmt_err --> wait_not_ok,
      ok & rd & !pdrd --> prd1 eofout_d=0,
            --> rdwait;

state prd1 = 0000
#        {let pldec = !tmt_err & !eofout_q;}
        {let pld1 = !tmt_err & !eofout_q;
        let pld2 = !tmt_err & !eofout_q & !flg8;}
      tmt_err --> wait_not_ok,
      done & !pdrd & eofin & !eofout_q --> wait_not_ok,
      !pdrd & eofin & !flg8 & !eofout_q --> prd2 eofout_d=0,
      !pdrd & eofin & flg8 & !eofout_q --> etbt1 eofout_d=0,
      !pdrd | sb_err --> prd1 eofout_d=0,
            --> prd1 eofout_d=1;

state prd2 = 0100
#        {let pldec = !tmt_err;}
        {let pld3 = !tmt_err;
        let pld4 = !tmt_err;}
      tmt_err --> wait_not_ok,
      done & eofin --> wait_not_ok,
      prco & eofin --> prd1 eofout_d=1,
      !prco & eofin --> prd1 eofout_d=0,
           --> prd2 eofout_d=0;

state etbt1 = 0011
#        {let pldec = !tmt_err;}
        {let pld2 = !tmt_err;}
      tmt_err --> wait_not_ok,
      done & eofin --> wait_not_ok,
      eofin --> etbt2 eofout_d=0,
            --> etbt1 eofout_d=0;
```

```
state etbt2 = 0001
#        {let pldec = !tmt_err;}
        {let pld3 = !tmt_err;}
      tmt_err --> wait_not_ok,
      done & eofin --> wait_not_ok,
      eofin --> etbt3 eofout_d=0,
             --> etbt2 eofout_d=0;

state etbt3 = 0110
#        {let pldec = !tmt_err;}
      tmt_err --> wait_not_ok,
      done & eofin --> wait_not_ok pld4=1,
      prco & eofin --> prd1 pld4=1 eofout_d=1,
      eofin --> prd1 pld4=1 eofout_d=0,
             --> etbt3 eofout_d=0 pld4=1;

state wait_not_ok
      !ok --> rdwait,
          --> wait_not_ok;
end

# 11/27/90 (faust)
#   - changed wr input to rd

# 12/20/90 (faust)
#   - eliminated pinc output which was identical
#     to pld4 output, screwing up e2v translator

# 2/12/91 (faust)
#   - added sbus_err (sbus error) and tmt_err (cam timeout
#      error) inputs.

# 2/27/91 (faust)
#   - deleted entime output.  Logic for enabling timeout
#       timer is now in the tmctr block.

# 7/22/91 (faust)
#   - fixed bug which allowed rdfsm to start prematurely
#       after a timeout.

# 9/4/91 (faust)
#   - modified to remove eofin as a qualifier of pldec; this
#     fixes a timing problem.  Removed pldec as an output and
#     generated it externally because logic synthesizer produced
#     unusable result (see notes.txt).

# 9/10/91 (faust)
#   - modified to remove eofin as a qualifier of pld1, pld2, pld3,
#     and pld4; this fixes a timing problem.

# 11/14/91 (faust)
#   - Changed timing of done to be one clock earlier.
#     Changed eofout timing to be one clock early so it can
#     be used on external flipflop.

# 11/25/91 (faust)
#   - Added input prco to allow eofout to be set when the read
#     data buffer is filled.
```

```
#     Added eofout_q input and changed output eofout to eofout_d
#     to reflect new use of this output.

# 12/11/91 (faust)
#   - Fixed byte mode bug (pld4 not generated when going to
#     wait_not_ok state).

# 1/13/92 (faust)
#   - In state etbt3, removed eofin as a qualifier of pld4.
```

[la]ps_sync by nhm on 16-Feb-93 at 12:21 A.M.

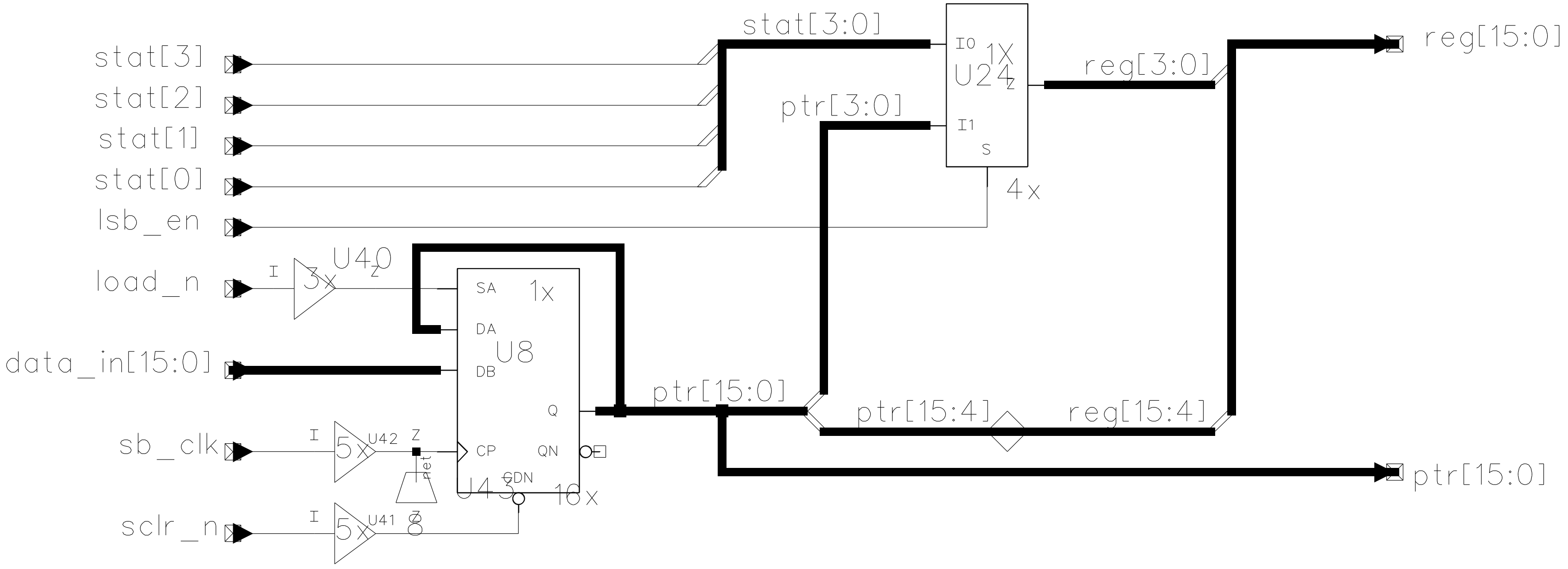


Controller chips

ps_sync.ps

[la]ptr_reg by nhm on 16-Feb-93 at 12:21 A.M.

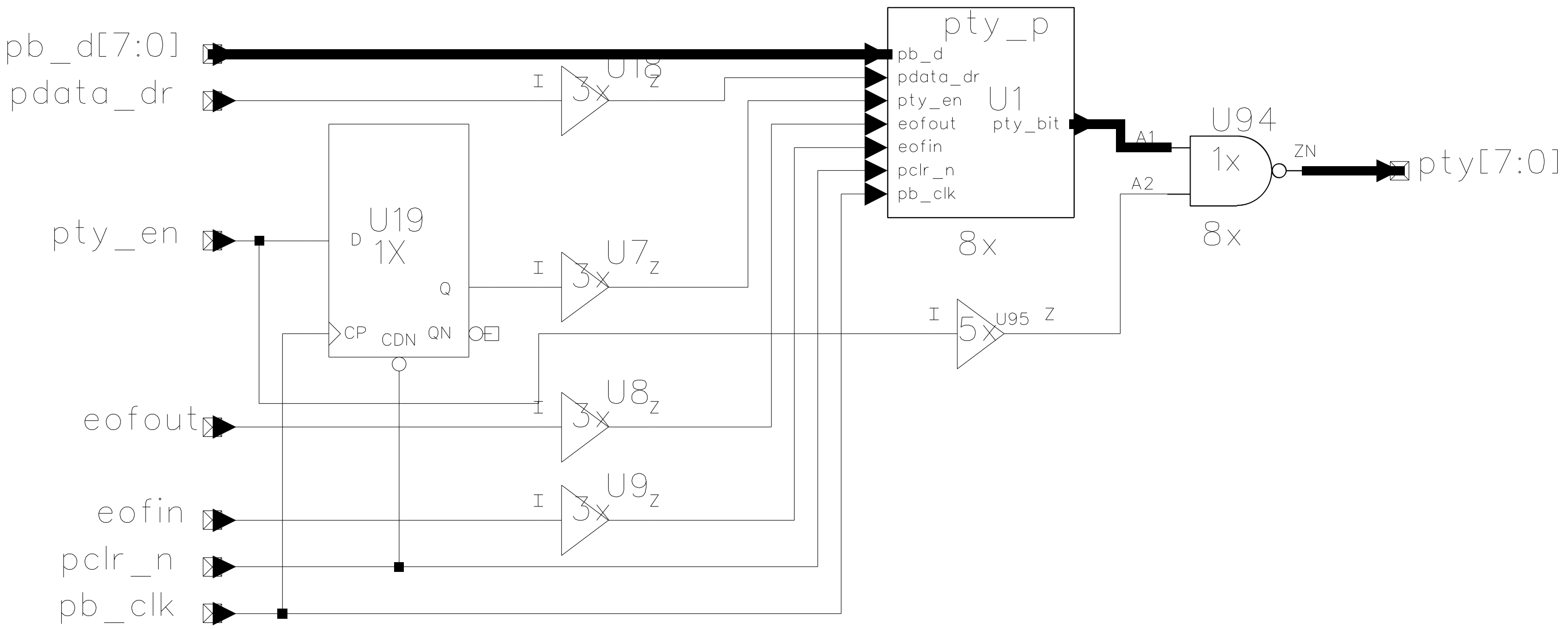
ptr_reg.ps



Controller chips

```
#cell2 * pty sm * 15 any 0 v8r1.8
# "7-May-91 GMT" "17:04:47 GMT" "7-May-91 GMT" "17:04:47 GMT" faust * .
sm pty;

# Finite State Machine file for the Parity Generation Hardware
# AUTHOR : David Harnanan
# FILENAME : /im/cam8/interface/pty.sm
# DATE : 16 May 1990       REVISED : 5/7/91

# 2/28/91 (faust)
#   - changed name drpty to pty_en.
#   - changed name parbit to pty_bit.
#   - changed state names to pty1 and pty0.
#   - changed the output to an rs ff output.
#   - added pdata_dr input and changed fsm to reflect
#     inverted sense of eofin on reads.

# 5/7/91 (faust)
#   - changed name rst_n to pclr_n.

clock pb_clk 25;

reset !pclr_n --> pty1;

inputs  eofin eofout pty_en pb_d pdata_dr;

rs=1 outputs pty_bit;

let ptydone = !eofin & !eofout & pty_en;

state pty1
        ptydone --> pty1,

        # flip parity if data is a one, is valid and is accepted
        (pdata_dr != eofin) & !eofout & !pty_en & pb_d --> pty0 pty_bit=0,
                --> pty1;

state pty0
        ptydone --> pty1 pty_bit=1,

        # flip parity if data is a one, is valid and is accepted
        (pdata_dr != eofin) & !eofout & !pty_en & pb_d --> pty1 pty_bit=1,
                --> pty0;

end
```

[la]ptygen8 by nhm on 16-Feb-93 at 12:21 A.M.



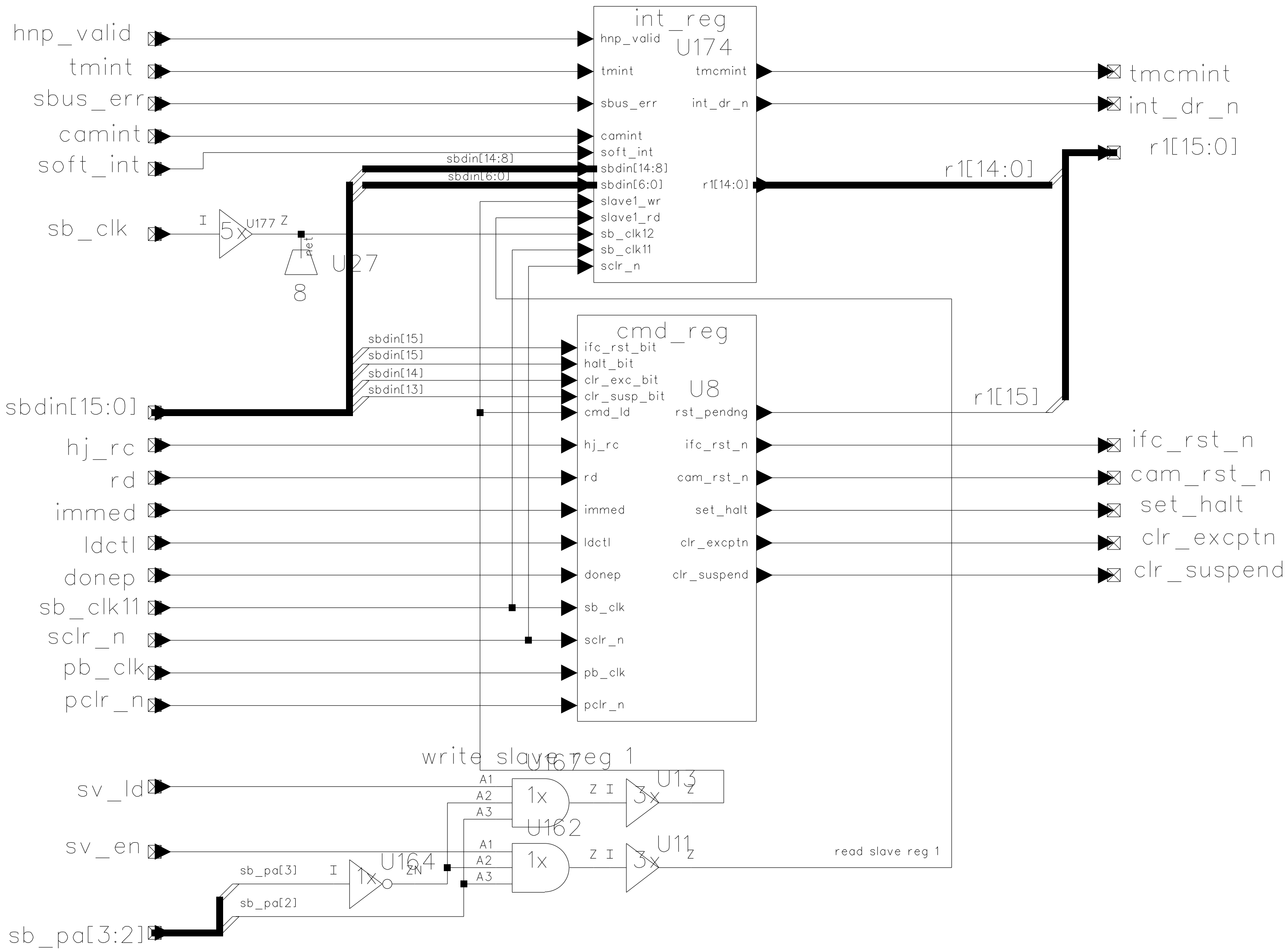
ptygen8.ps

Controller chips

```
#cell2 * pwrfsm sm * 34 any 0 v8r2.2
# "2-Jan-92 GMT" "13:05:05 GMT" "2-Jan-92 GMT" "13:05:05 GMT" faust * .
sm pwrfsm;

# Finite State Machine file for the CAM Write FSM of the PFSM System
# AUTHOR : David Harnanan
# FILENAME : /im/cam8/interface/pwrfsm.sm
# DATE : 14 April 1990       REVISED : 1/2/92

clock pb_clk 25;

reset !rst_n --> wrwait;

inputs    pdav ok rd prco immed eofin done error;

outputs
        pdata_en        # pbus data enable
        pdata2_en       # pbus data enable (sbus lsbs), gated
                        #  with !eofin in dataregs chip
        e_pdata2_en     # pbus data enable (sbus lsbs), gated
                        #  with eofin in dataregs chip
        pldec           # decrement word count
        pinc            # bump data register ptr
        pcwr            # clear data available flag
        eofout=1;       # pbus handshake output

# Note that the timeout counter is enabled between two enables
# of the PBus.  If this period exceeds 256 clock cycles, there
# is a bus timeout.  Also note that the timeout counter is cleared
# when an enable occurs -- this restarts the counting process.
# The counter is disabled when the system must wait on the SBus
# to fill the Datain registers.

state wrwait
        error --> wrwait pcwr=1,
        ok & !rd & pdav --> pwr1 pdata_en=1 eofout=0,
            --> wrwait;

state pwr1
          {let pdata_en = !error;
          let pdata2_en = !error & pdav;
          let pldec = !error & pdav;}
        error --> wrwait pcwr=1,
        done & !eofin & pdav --> last_word,
        !eofin & pdav --> pwr2 eofout=0,
        eofin & pdav --> pwr1 eofout=0,
                   --> wrwait;

state pwr2
          {let pldec = !error;
          let pdata_en = !error;
          let e_pdata2_en = !error;}
        error --> wrwait pcwr=1,
        done & !eofin --> last_word,
        !eofin & immed --> pwr1 eofout=0,
        !eofin & !prco --> pwr1 pinc=1 eofout=0,
        !eofin & prco --> pwr1 pinc=1 pcwr=1,
                  --> pwr2 eofout=0;
```

```
state last_word
       !ok --> wrwait,
           --> last_word;

end

# 11/27/90 (faust)
#   - changed wr input to rd
#   - decoded noop internally

# 2/12/91 (faust)
#   - added error input (sbus or cam timeout).

# 2/19/91 (faust)
#   - added entim=1 to default condition of state
#     pwr2 and removed entim=0 from lines which
#     output clrtim=1.

# 2/27/91 (faust)
#   - deleted entim output.  Logic for enabling timeout
#       timer is now in the tmctr block.

# 2/28/91 (faust)
#   - changed signal name pdrdata to pdata_en.
#   - changed signal name pdr2 to pdata2_en.
#   - made timing of pdata_en and pdr2 one clock earlier
#     so it could be clocked into a ff in the dataregs chip.

# 3/6/91 (faust)
#   - deleted clrtim output.

# 9/3/91 (faust)
#   - modified to remove eofin as a qualifier of pldec; this
#     fixes a timing problem.

# 9/12/91 (faust)
#   - modified to remove eofin as a qualifier of
#     pdata2_en; this fixes a timing problem.
#     Added output e_pdata2_en (enabled by eofin).

# 11/20/91 (faust)
#   - Changed timing of done to be one clock earlier.
#     Changed eofout timing to be one clock early so it can
#     be used on external flipflop.

# 1/2/92 (faust)
#   - Modified to eliminate pldec during noop (this is now
#     done by the pctlfsm).
```

[la]reg_1 by nhm on 16-Feb-93 at 12:21 A.M.

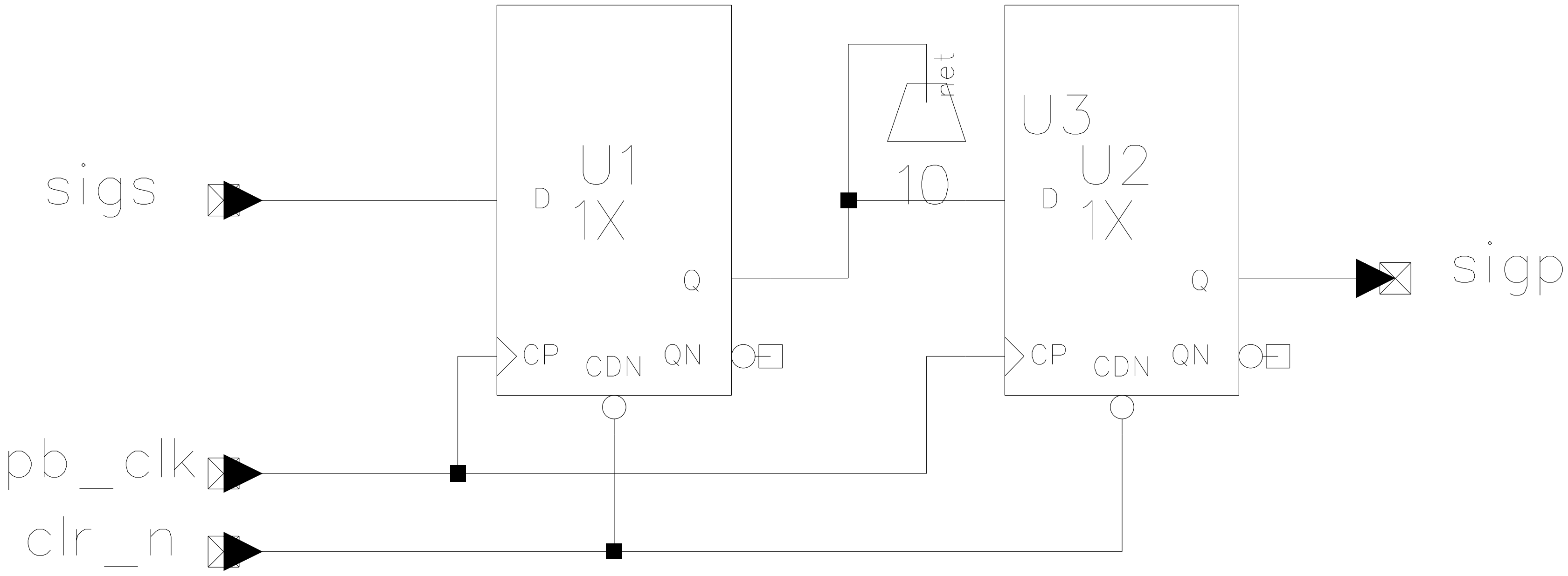


Controller chips

reg_1.ps

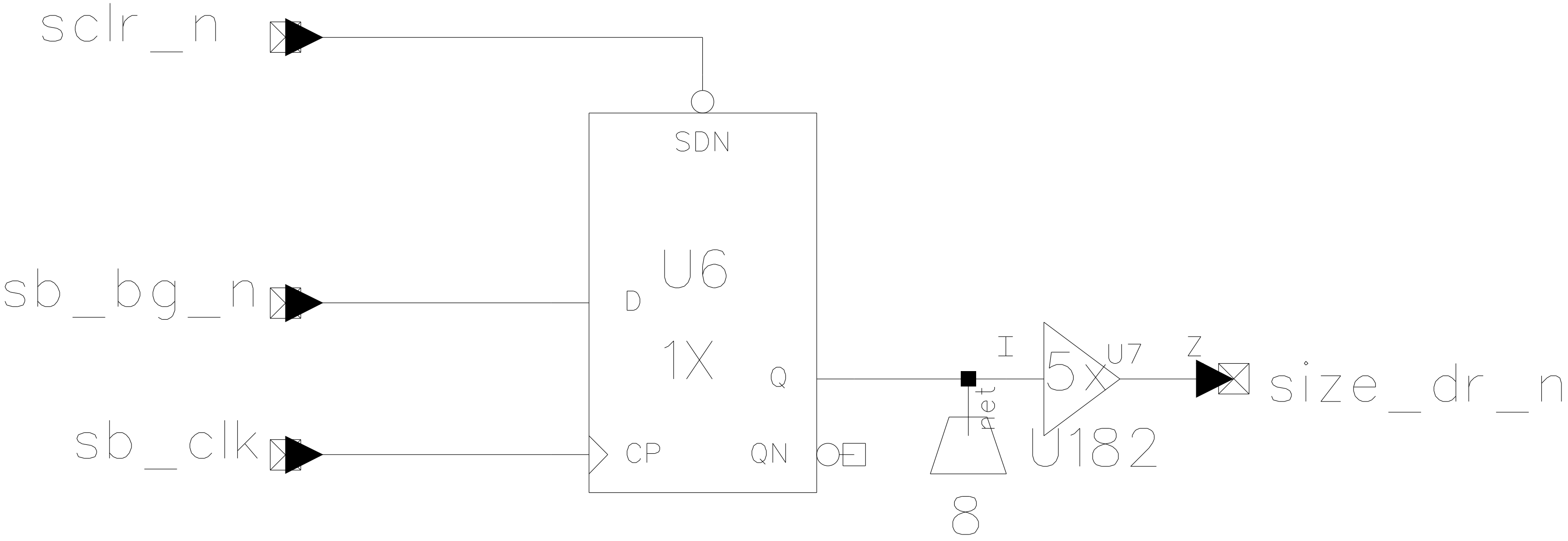
[la]s_psyn by nhm on 16-Feb-93 at 12:21 A.M.

s_psyn.ps



Controller chips

[la]sb_drv by nhm on 16-Feb-93 at 12:21 A.M.

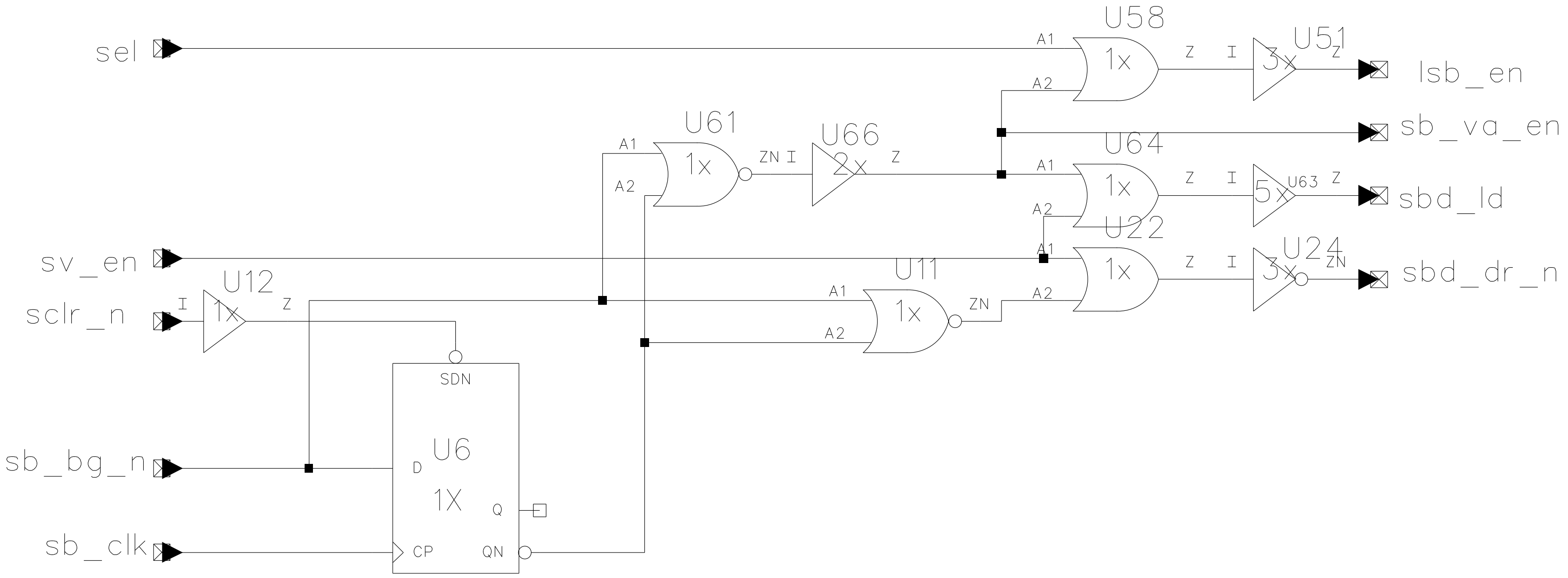


Controller chips

sb_drv.ps

[la]sb_va_gen by nhm on 16-Feb-93 at 12:21 A.M.

sb_va_gen.ps

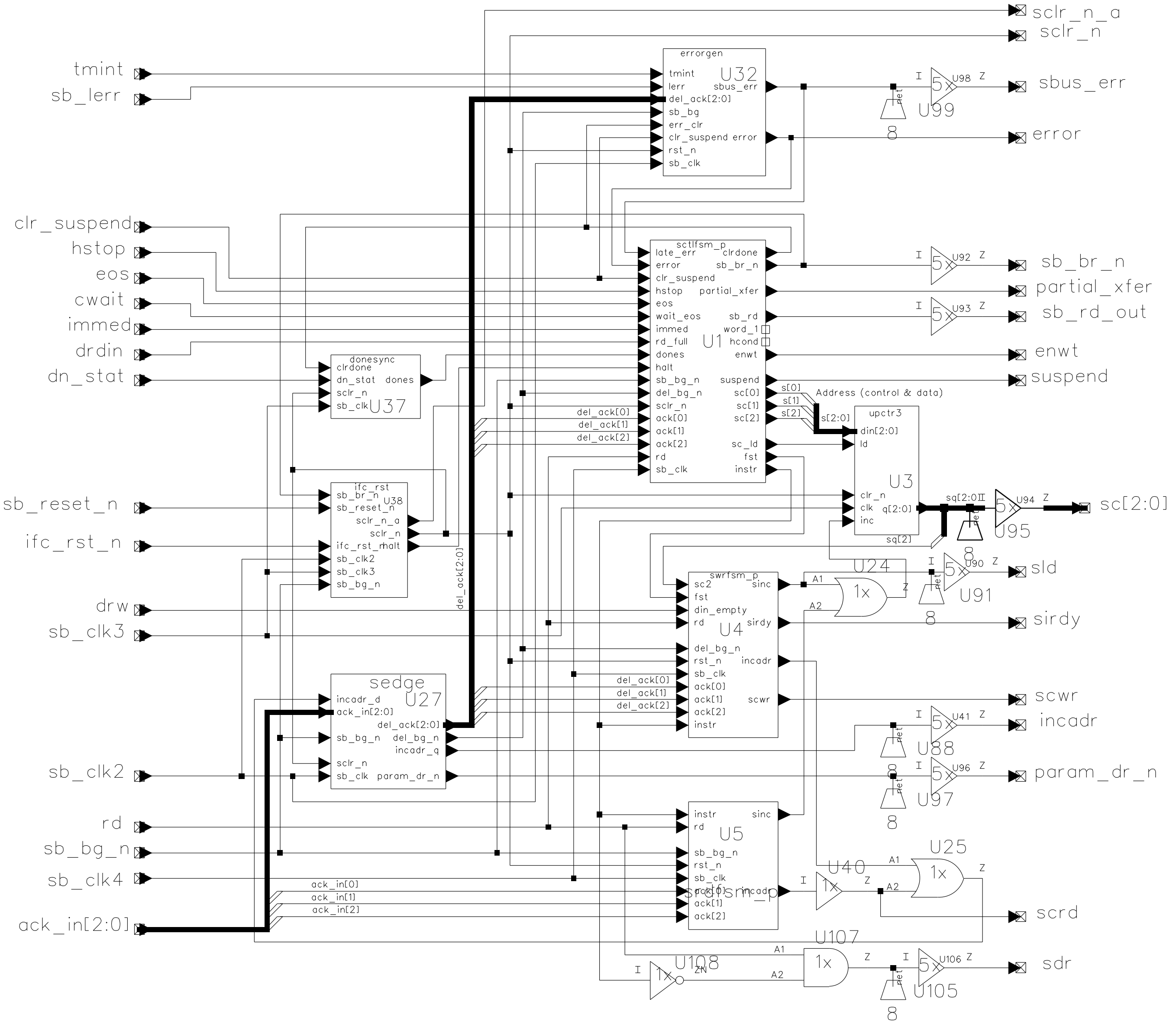


Controller chips

```
#cell2 * sbugfixfsm sm * 12 any 0 v8r1.8
# "24-Apr-91 GMT" "11:12:53 GMT" "24-Apr-91 GMT" "11:12:53 GMT" faust * .
sm sbugfixfsm;

# Finite State Machine file to fix SBus ERR ACK bug
# AUTHOR : David Harnanan
# FILENAME : /im/cam8/interface/sbugfixfsm.sm
# DATE : 5 July 1990       REVISED : 4/24/91

# 4/24/91 faust
#  - changed signal name rst to sb_reset_n
#  - changed signal name sb_bg to sb_bg_n

clock sb_clk 20;
reset !sb_reset_n --> bugidle;

inputs  ack[2:0] sb_bg_n;

outputs   ackout0=1 ackdr;

let errack = (ack = 110);

state bugidle
     errack & !sb_bg_n --> bugwt1,    # Detect an ERR acknowledgment while a master
         --> bugidle;

state bugwt1
     sb_bg_n --> bugidle,           # if Grant* deasserted, system is not hung
           --> bugwt2;            # otherwise, wait some more ...

state bugwt2
     sb_bg_n --> bugidle,           # if Grant* deasserted, system is not hung
           --> bugwt3;            # otherwise, wait some more ...

state bugwt3
     sb_bg_n --> bugidle,           # if Grant* deasserted, system is not hung
           --> bugwt4;            # otherwise, wait some more ...

state bugwt4
     sb_bg_n --> bugidle,           # if Grant* deasserted, system is not hung
           --> bugwt5;            # otherwise, wait some more ...

state bugwt5
     sb_bg_n --> bugidle,           # if Grant* deasserted, system is not hung
           --> bugwt6;            # otherwise, wait some more ...

state bugwt6
     sb_bg_n --> bugidle,                 # if Grant* deasserted, system is not hung
           --> bugfix ackout0=0 ackdr=1;    # assume by now that system is hung ...

state bugfix
     sb_bg_n --> bugidle ackdr=1,       # system unhung, drive ack high as required
           --> bugfix ackout0=0 ackdr=1;  # assert err ack until Grant*
                                          # is deasserted

end
```

[la]sbus_ctl by nhm on 16-Feb-93 at 12:21 A.M.

sbus_ctl.ps



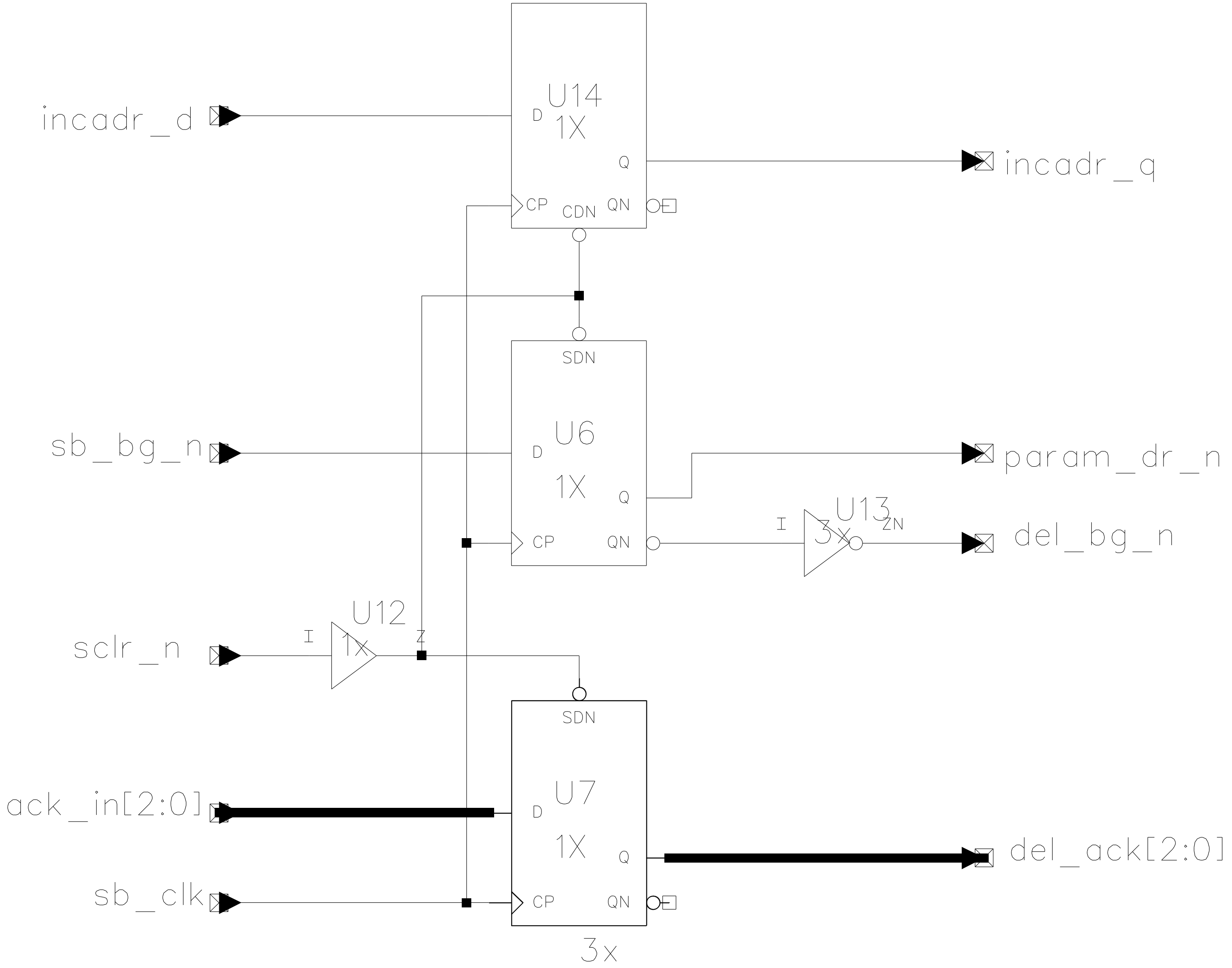
Controller chips

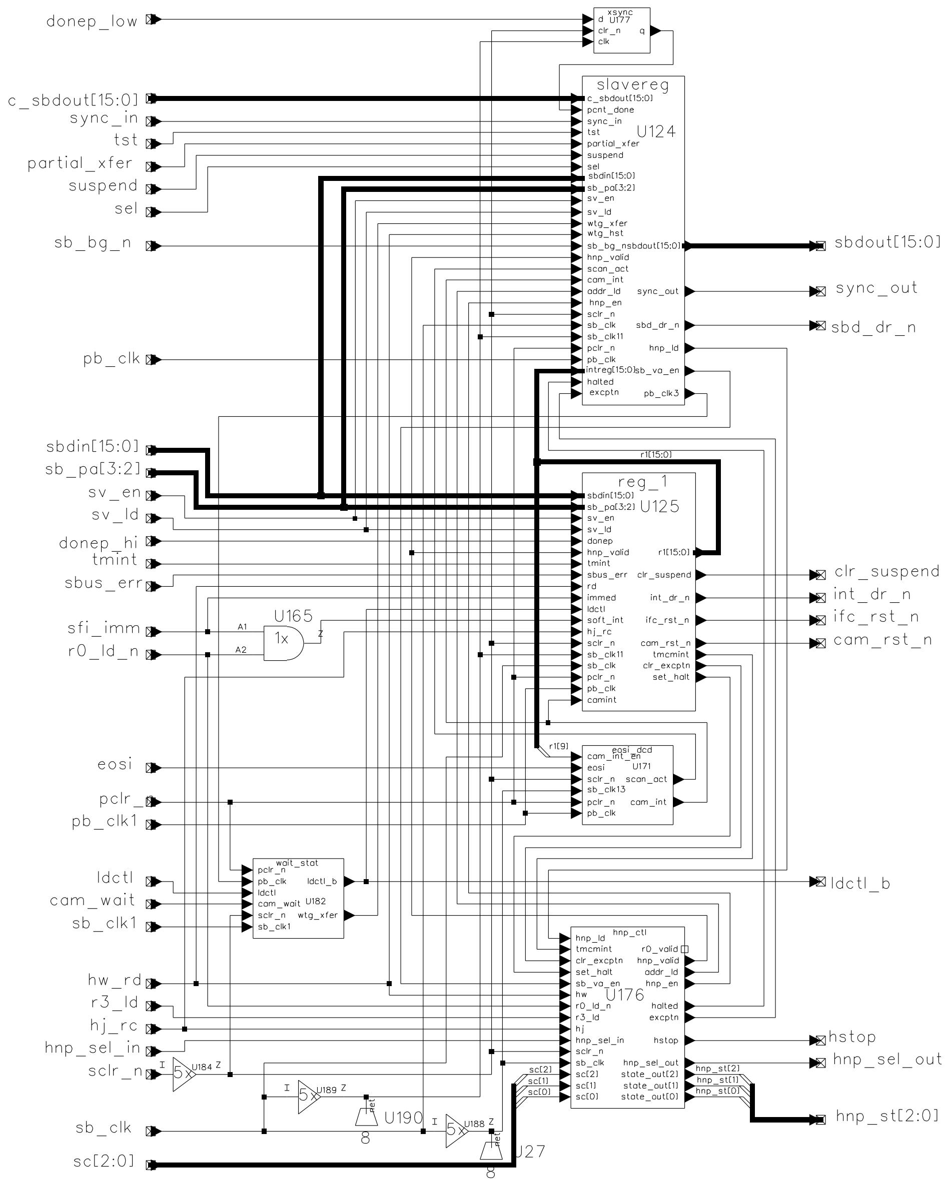
```
#cell2 * sctlfsm sm * 69 any 0 v8r1.8
# "12-Jun-91 GMT" "11:11:38 GMT" "12-Jun-91 GMT" "11:11:38 GMT" faust * .
sm sctlfsm;

# Finite State Machine file for the Control FSM of the SFSM System
# AUTHOR : David Harnanan
# FILENAME : /im/cam8/interface/sctlfsm.sm
# DATE : 19 April 1990       REVISED : 6/13/91

clock sb_clk 20;

reset !rst_n --> init;

inputs
        rd              # read mode bit (from ctl reg 0)
        del_bg_n        # bus grant, dly'd one clock
        rd_full         # read data regs full
        error           # cam timeout/sbus error
        immed           # immediate data bit
        dones           # cam xfer done
        wait_eos        # wait for eos bit
        eos             # end of scan status
        hstop           # stop signal from ctlsvchip
        ack[2:0]        # sbus ack signals
        sb_bg_n            # sbus grant signal
        late_err        # sbus late_err
        clr_suspend     # clear suspended state
        halt;           # prevent further bus requests

outputs
        sc_ld           # load ctl/data reg pointer
        clrdone         # clear done & error status
        sc[2:0];        # data input to ctl/data reg pointer

rs=0 outputs
        fst             #
        enwt            # enable wait for cam eos
        sb_rd           # controls sbus data direction
        hcond           # condition which requires waiting for host
        instr           # current burst is instruction, not data
        suspend         # interface operation is suspended because
                        #   of error acknowledgement
        partial_xfer    # an error ack or late error has caused
                        #   a partial transfer of data to cam
        word_1;         # 1 during first word of burst
rs=1 outputs
        sb_br_n;        # Sbus request


let   rerun  = (ack = 100);
let   err_ack  = (ack = 110);
let   norm_ack  = (ack = 011);

state init
      --> waitst hcond=1;

state waitst
        !partial_xfer & hcond & !hstop --> instrld
                clrdone=1 hcond=0 instr=1 sb_rd=1 fst=1
```

```
                sc_ld=1 sc='d3,
        !partial_xfer & !hcond & wait_eos & eos & enwt -->
                mainprc enwt=0,
        clr_suspend --> waitst partial_xfer=0 suspend=0,
                --> waitst;

state instrld
     {let enwt=1;}
     !sb_bg_n & !sb_br_n --> instrld sb_br_n=1,
     !del_bg_n --> instrprc sc_ld=1,
     halt --> instrld,
            --> instrld sb_br_n=0 word_1=1 ;

state instrprc
     rerun --> rrnwt1,
     err_ack --> rrnwt1 suspend=1,
     norm_ack --> instrprc word_1=0,
     !del_bg_n --> instrprc,
               --> mainprc sc_ld=1 sc=1 instr=0;

state rrnwt1
     !suspend & del_bg_n --> instrld sc_ld=1 sc='d3,
     clr_suspend --> rrnwt1 suspend=0,
              --> rrnwt1;

state mainprc
     !del_bg_n & !rd --> wrprc sc_ld=1 sc='d4 word_1=1,
     !del_bg_n & rd --> rdprc sc_ld=1 sc='d4,
     !sb_bg_n & !sb_br_n --> mainprc sb_br_n=1,
     halt --> mainprc,
     late_err & sb_br_n --> waitst hcond=1 partial_xfer=1,
     late_err & !sb_br_n --> mainprc partial_xfer=1,
     (partial_xfer | error) & sb_br_n --> waitst hcond=1,
     (partial_xfer | error) & !sb_br_n --> mainprc,
     wait_eos & enwt --> waitst hcond=0,

     # xmt the last read from CAM
     !immed & rd & dones & fst --> mainprc sb_br_n=0 sb_rd=0,

     dones & sb_br_n --> waitst hcond=1 enwt=0,

     !immed & !rd --> mainprc sb_br_n=0,
     !immed & rd & (rd_full | dones) --> mainprc sb_br_n=0 sb_rd=0,
             --> mainprc;

state rdprc
     rerun --> rrnwt2,
     err_ack --> rrnwt2 suspend=1,
     del_bg_n & dones --> mainprc fst=0 sc_ld=1 sc=1,
     del_bg_n --> mainprc sc_ld=1 sc=1,
              --> rdprc;

state wrprc
     rerun --> rrnwt2,
     err_ack & word_1 --> rrnwt2 suspend=1,
     err_ack --> wrprc word_1=0 partial_xfer=1 suspend=1,
     norm_ack --> wrprc word_1=0,
     del_bg_n --> mainprc fst=0 sc_ld=1 sc=1,
           --> wrprc;
```

```
state rrnwt2
     !suspend & del_bg_n --> mainprc sc_ld=1 sc=1,
     clr_suspend --> rrnwt2 suspend=0,
           --> rrnwt2;

end

# 11/27/90 (faust)
#   - changed wr input to rd
#   - decoded noop internally

# 12/20/90 (faust)
#   - eliminated errint output which was identical
#     to clrdone output, screwing up e2v translator

# 1/29/91 (faust)
#   - changed state mainprc to prevent release of
#     bus request if done becomes true.  Also changed
#     br output to be a flipflop.

# 1/30/91 (faust)
#   - in mainprc, the fix to prevent premature release
#     of br was bad, redid it.

# 1/30/91 (faust)
#   - redefined the done signal to not include noop; this
#     prevents premature exit from the mainprc state.

# 2/19/91 (faust)
#   - Added sbus_err input.  Modified the fsm to do
#       the following:
#
#       sbus_err:  complete any burst transfer in progress (if
#                  an error acknowledgement occurs, the Sbus
#                  controller will abort the transfer by
#                  deasserting bus grant), then go to waitst if
#                  cam read not in progress, else wait for dones,
#                  then go to waitst.
#       error:     same as sbus_err, but don't wait for done if
#                  cam read in progress.

# 3/7/91 (faust)
#   - Added early_done input.  Changed mainprc to not do another
#     bus request if cam write and early done are true.

# 3/26/91 (faust)
#   - Changed signal name ld to sc_ld.
#   - Changed signal name br to sb_br_n.
#   - Changed signal name drdin to rd_full.
#   - Changed signal name busdir to sb_rd.
#   - Changed signal name cwait to wait_eos.
#   - Changed signal name sbus_err to late_err.
#   - Deleted early_done input.
#   - Added rs=0 output suspend
#   - Added rs=0 output word_1
#   - Added rs=0 output abort
#   - Added input clr_suspend
#   - Modified to treat error acknowledges like reruns unless
```

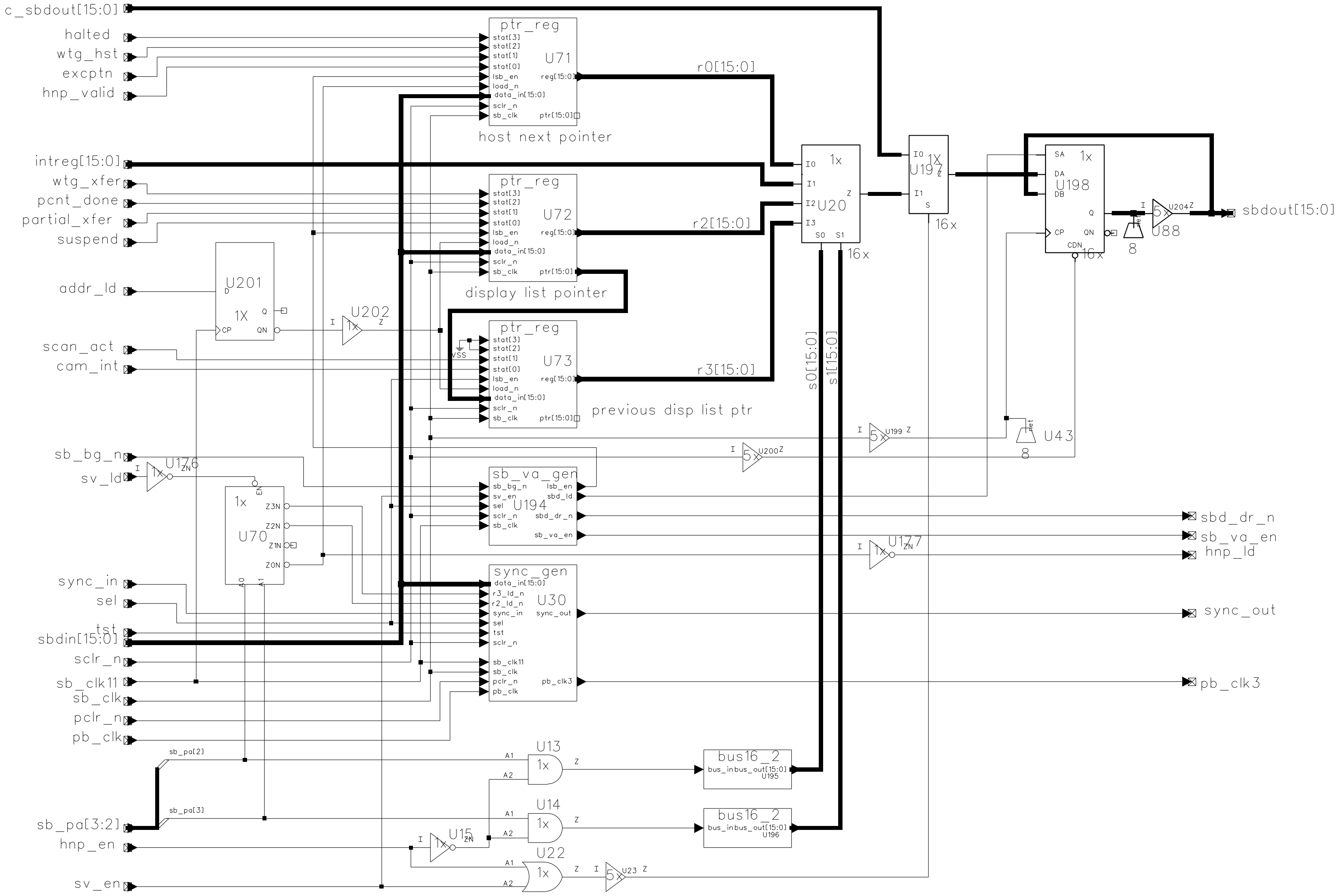
```
#     it is not the first one in a burst.  An error acknowledge
#     after the first word or a late error is treated as a
#     partial transfer.

# 4/3/91 (faust)
#   - Changed name of state wrprc1 to wrprc.
#   - Fixed bugs in rdprc and wrprc states (clerical error).

# 4/4/91 (faust)
#   - Fixed bug in mainprc; word_1 wasn't set to 1 before going
#     to wrprc state.
#   - Changed name abort to partial_xfer

# 4/18/91 (faust)
#   - Added halt input to indicate an impending reset; this
#     is used to prevent bus request from being set.

# 5/1/91 (faust)
#   - Changed signal name bg_n to sb_bg_n.

# 6/12/91 (faust)
#   - Changed mainprc so that a late error which occurs while
#     bus request is asserted will not be serviced until the
#     bus is granted.

# 6/13/91 (faust)
#   - Fixed bug in waitst--suspend not cleared by clr_suspend.
```


incadr_d
U14
1X
D
Q
CP
CDN
QN
incadr_q
SDN
U6
1X
D
Q
CP
QN
sb_bg_n
param_dr_n
U13
3x
I
ZN
del_bg_n
U12
1x
I
Z
sclr_n
SDN
U7
1X
D
Q
CP
QN
ack_in[2:0]
del_ack[2:0]
sb_clk
3x



Controller chips

[la]slavereg by nhm on 16-Feb-93 at 12:22 A.M.



Controller chips

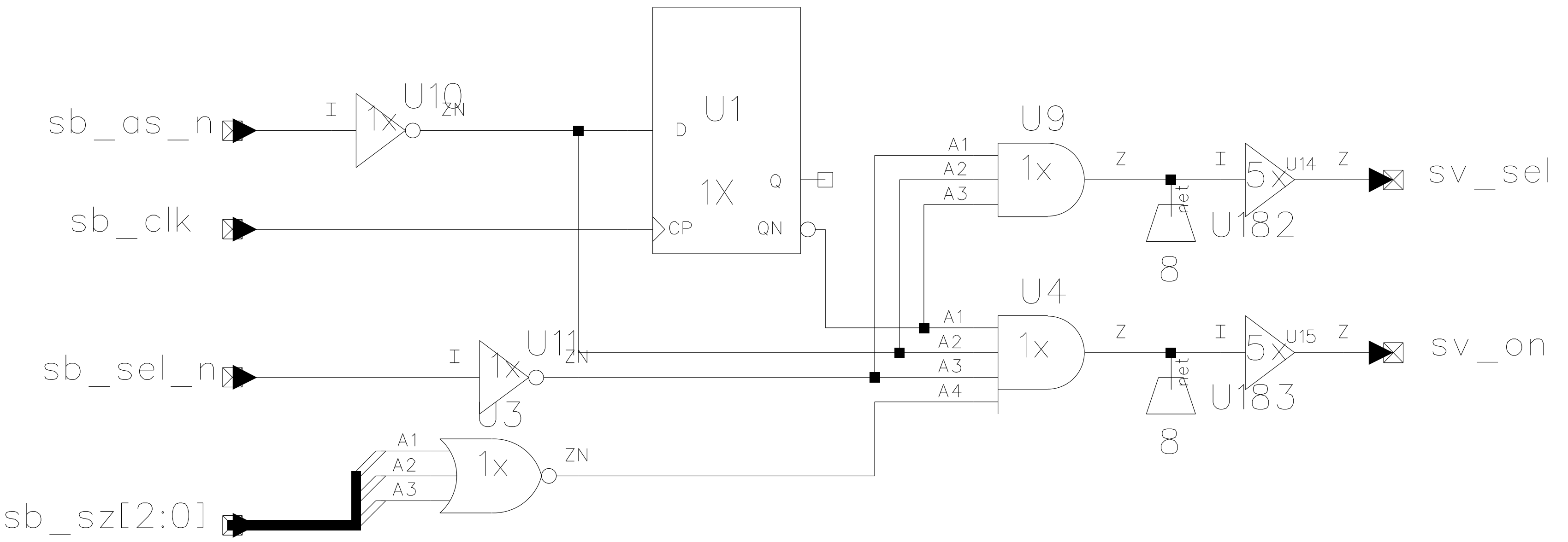
slavereg.ps

```
#cell2 * srdfsm sm * 20 any 0 v8r2.5
# "29-Apr-92 GMT" "14:12:35 GMT" "29-Apr-92 GMT" "14:12:35 GMT" faust * .
sm srdfsm;

# Finite State Machine file for the CAM Read FSM of the SFSM System
# AUTHOR : David Harnanan
# FILENAME : /im/cam8/interface/srdfsm.sm
# DATE : 10 May 1990       REVISED : 4/29/92

# 11/27/90 (faust)
#   - changed wr input to rd.

# 12/18/90 (faust)
#   - eliminated scwr output which was identical
#     to incadr output, screwing up e2v translator

# 2/19/91 (faust)
#   - eliminated error input and modified the fsm
#     to abort if bus grant is released.  Also
#     changed name from sb_bg to del_bg_n.

# 3/26/91 (faust)
#   - changed filename to srdfsm.sm

# 5/1/91 (faust)
#   - changed signal name del_bg_n to sb_bg_n

# 10/3/91 (faust)
#   - changed function of sdr to be a d input
#     to a ff (in dataregs) to help data timing

# 4/29/92 (faust)
#   - removed sdr output--this is now done by an external gate.

clock sb_clk 20;

reset !rst_n --> start;

inputs sb_bg_n ack[2:0] rd instr;

outputs  sinc incadr;

let ak     = (ack = 011);
let errack = (ack = 110);
let rerun  = (ack = 100);

state start
      !sb_bg_n&rd&!instr --> camrd0,
            --> start;

state camrd0
      sb_bg_n --> start,    # abort if no bus grant
            --> camrd1;     # dummy state to synch with par drive FSM

state camrd1
      rerun --> camrd5,
      errack | sb_bg_n --> start,
      ak --> camrd2 sinc=1,
          --> camrd1;
```

```
state camrd2
      errack | sb_bg_n --> start,
      ak --> camrd3 sinc=1,
          --> camrd2;

 state camrd3
      errack | sb_bg_n --> start,
      ak --> camrd4 sinc=1,
          --> camrd3;

state camrd4
      errack | sb_bg_n --> start,
      ak --> camrd5 incadr=1,
          --> camrd4;

state camrd5
      sb_bg_n --> start,
            --> camrd5;

end
```

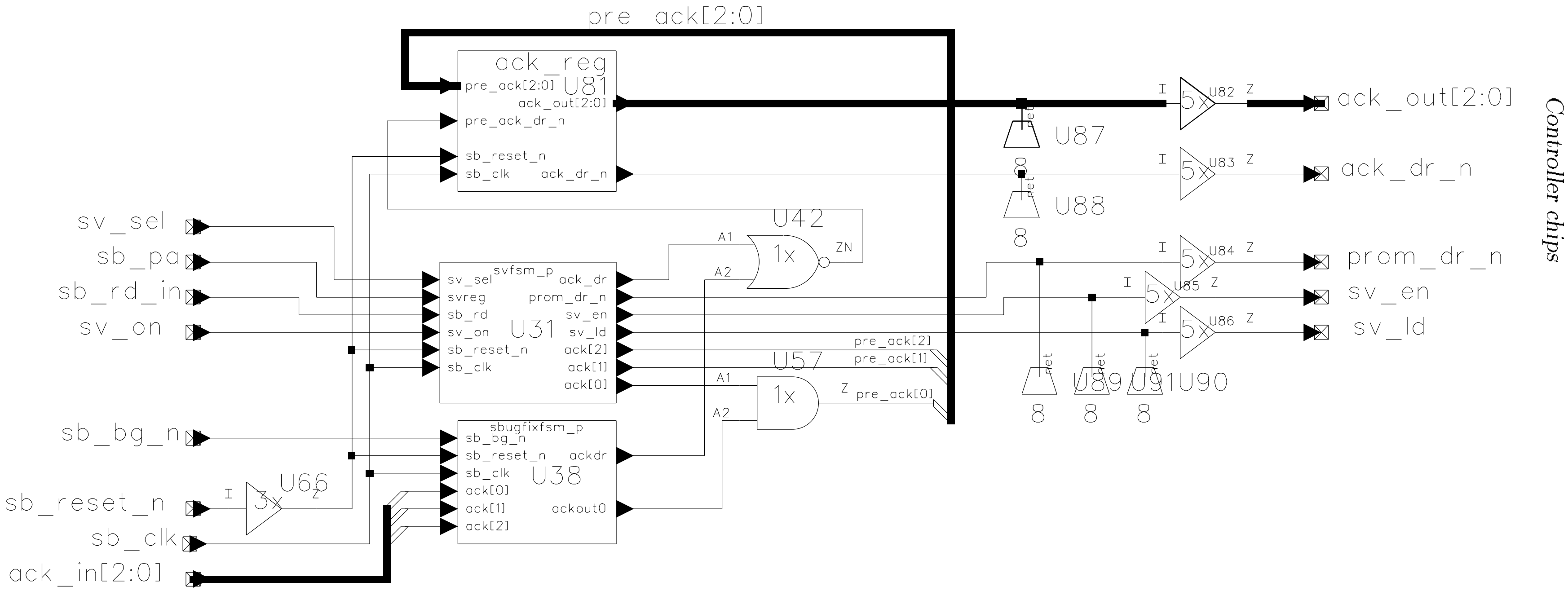
[la]sv_oncircuit by nhm on 16-Feb-93 at 12:22 A.M.

sv_oncircuit.ps



Controller chips

[la]sv_sbug by nhm on 16-Feb-93 at 12:22 A.M.



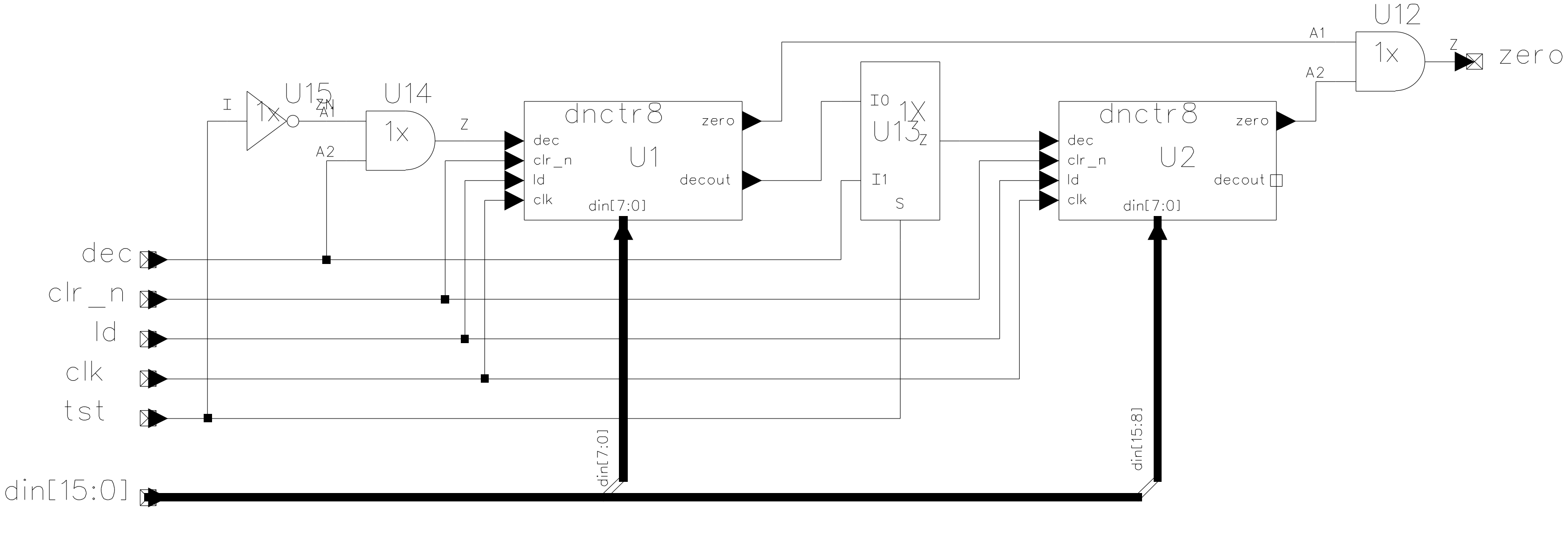
sv_sbug.ps

```
#cell2 * svfsm sm * 34 any 0 v8r2.2
# "8-Oct-91 GMT" "9:03:46 GMT" "8-Oct-91 GMT" "9:03:46 GMT" faust * .
sm svfsm;

# Finite State Machine file for the Slave FSM of the Interface Board
# AUTHOR : David Harnanan
# FILENAME : /im/cam8/interface/svfsm.sm
# DATE : 26 June 1990       REVISED : 10/8/91

# 12/4/90 - df
#   - changed prmwait state to go to svidle unconditionally
#   - deleted pmctr and related signals from fsmchip

# 12/18/90 - df
#   - added error acknowledge response
#   - changed name "on" to "sv_on"

# 4/17/91 - df
#   - fixed error acknowledge response

# 4/24/91 - df
#   - changed signal name sclr_n to sb_reset_n

# 5/1/91 - df
#   - changed signal name svdr to sv_en
#   - changed signal name svld to sv_ld

# 5/7/91 - df
#   - changed signal name prmdr_n to prom_dr_n
#   - changed signal name ackdr to ack_dr

# 10/3/91 - df
#   - asserted sv_en one clock earlier to allow data to be
#     latched in ctlsvchip (for timing improvement).

# 10/8/91 - df
#   - changed timing of ack and ack_dr to be one clock earlier
#     and added ff to output (externally, to combine with output
#     of sbugfixfsm).

#
# NOTE: The Slave FSM and associated hardware is designed such that one of
#       the PA(27:0) lines of the SBus is used to distinguish between the
#       Slave Register locations and the Slave PROM address space.
#       This address line is wired to the svreg signal.
#       In order to write to the host next pointer or reset registers,
#       the Sparcstation must access the correct address of the Register
#       portion of the slave space.  Note that access to the Register
#       address space always responds with SBus word acknowledgments,
#       whereas access of the PROM address space uses bus sizing and
#       responds with byte acknowledgments.
#

clock sb_clk 20;

reset !sb_reset_n --> svidle;

inputs    sv_on sv_sel sb_rd svreg;    # svreg is just an address line of
                                           # PA(27:0)
```

```
outputs   ack[2:0]=111 ack_dr sv_ld sv_en;
rs=1 outputs  prom_dr_n;

state svidle
      sv_on & svreg --> sv1 ack[2]=0 ack_dr=1,
      sv_on & !svreg & sb_rd --> prmacc_1 prom_dr_n=0,
      sv_sel & !sv_on --> sverr1 ack[0]=0 ack_dr=1,
         --> svidle;

state sv1
      {let ack_dr=1;}
       sb_rd --> sv2 sv_en=1,
             --> sv2 sv_ld=1;

state sv2
       sb_rd --> svidle sv_en=1,
             --> svidle;

state prmacc_1
            --> prmacc_2;          # Drive PROM long enough to guarantee
                                   # a stable output
state prmacc_2
            --> prmacc_3;

state prmacc_3
            --> prmack ack[1]=0 ack_dr=1;

state prmack
       {let ack_dr =1;}
            --> prmdrive;

state prmdrive
            --> prmwait prom_dr_n=1;

state prmwait
            --> svidle;

state sverr1
      {let ack_dr=1;}            # error acknowledgement
             --> sverr2;

state sverr2
             --> svidle;
end
```

```
#cell2 * swrfsm sm * 25 any 0 v8r1.8
# "26-Mar-91 GMT" "12:30:05 GMT" "26-Mar-91 GMT" "12:30:05 GMT" faust * .
sm swrfsm;

# Finite State Machine file for the CAM Write FSM of the SFSM System
# AUTHOR : David Harnanan
# FILENAME : /im/cam8/interface/swrfsm.sm
# DATE : 10 May 1990       REVISED : 3/26/91

# 11/27/90 (faust)
#   - changed wr input to rd

# 12/18/90 (faust)
#   - eliminated sld output which was identical
#     to sinc output, screwing up e2v translator

# 2/19/91 (faust)
#   - removed error input; the fsm will "do the right
#     thing" by watching the bus grant signal.  Also
#     changed name from sb_bg to del_bg_n.

# 3/26/91 (faust)
#   - changed name from drw to din_empty

clock sb_clk 20;

reset !rst_n --> idle;

inputs del_bg_n ack[2:0] rd din_empty fst sc2 instr;

outputs  scwr sinc incadr sirdy;

let ak = (ack = 011);

state idle
        !del_bg_n & ak & instr --> camwr scwr=1 sinc=1,
        !del_bg_n & ak & !rd & (din_empty | fst) -->
                 camwr scwr=1 sinc=1,
        !del_bg_n & ak & !rd & !din_empty & !fst --> spill,
                --> idle;

state camwr
        # incadr only if data was loaded
        del_bg_n & !sc2 --> idle incadr=1,

        # sc is incremented to 000 by end
        del_bg_n & sc2 --> idle sirdy=1,

        # of 4 data transfers ...
        ak --> camwr sinc=1,
                --> camwr;

state spill
        del_bg_n --> idle,
                --> spill;

end
```

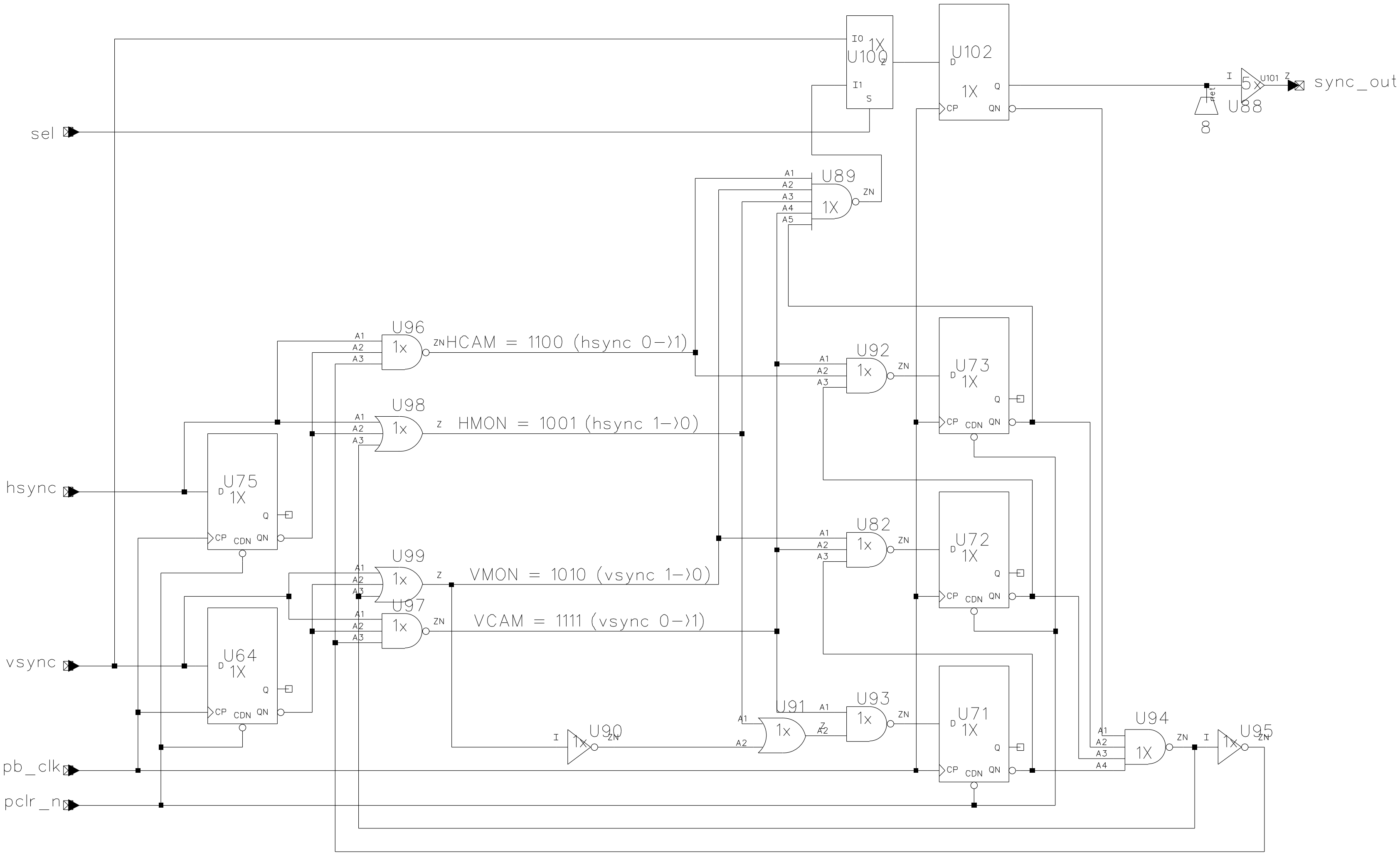
[la]sync_ctr by nhm on 16-Feb-93 at 12:22 A.M.



Controller chips

sync_ctr.ps

[la]sync_encode by nhm on 16-Feb-93 at 12:22 A.M.



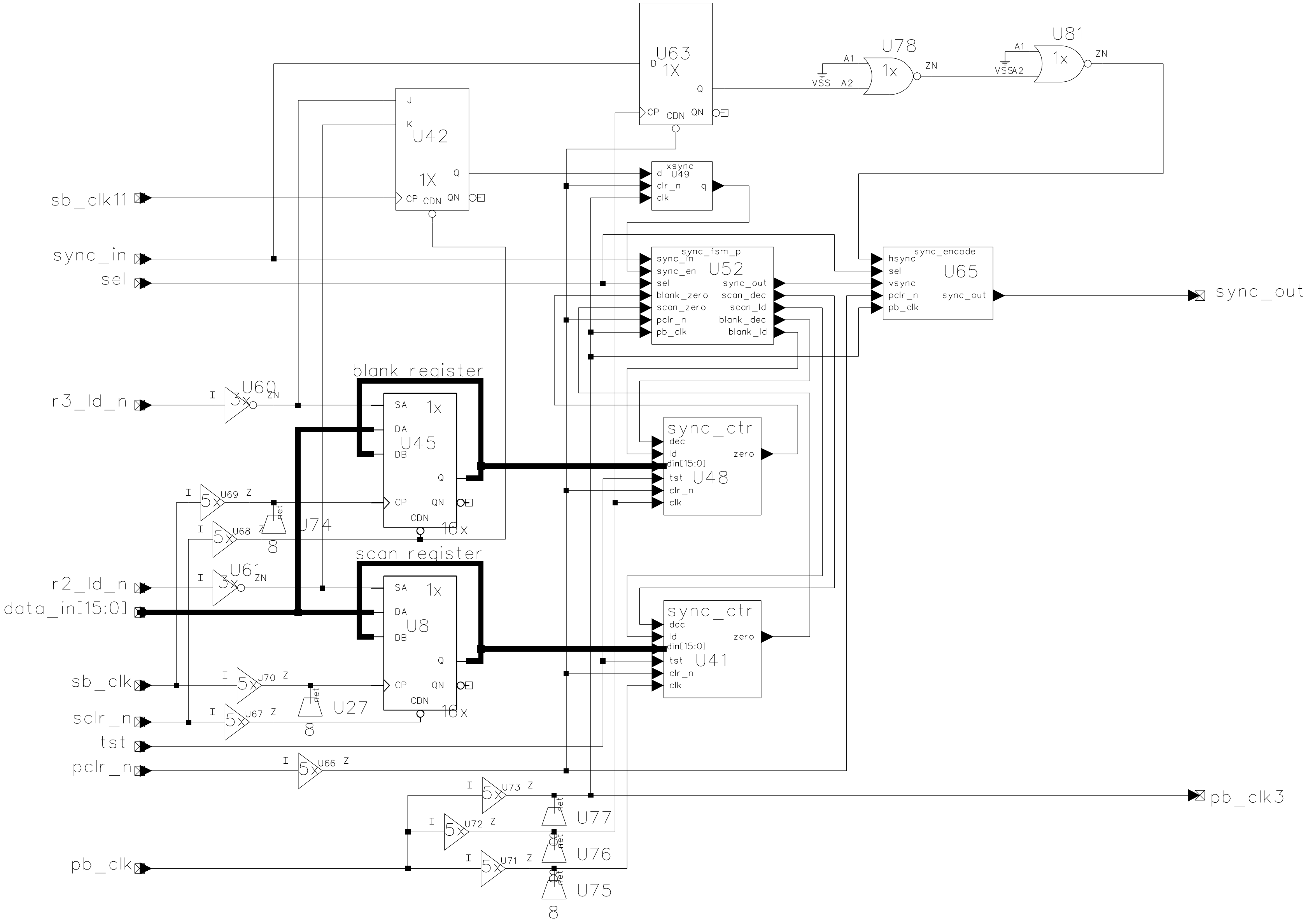
sync_encode.ps

Controller chips

```
#cell2 * sync_fsm sm * 16 any 1024 v8r2.2
# "3-Mar-92 GMT" "14:44:05 GMT" "3-Mar-92 GMT" "14:44:05 GMT" faust * .
sm sync_fsm;

# Finite State Machine file for the control of
#   the sync and blanking counters
# AUTHOR : Doug Faust
# FILENAME : /im/cam8/interface/sync_fsm.sm
# DATE : 28 Mar 1991       REVISED : 3/3/92

clock pb_clk 25;

reset !pclr_n --> idle_state ;

inputs
        sync_in         # sync input from LSB chip
        blank_zero      # blanking counter = 0
        scan_zero       # scan counter = 0
        sel             # wired to 0 for LSB chip, 1 for MSB chip
        sync_en;        # counters active (sync'd from sbus clock)

outputs
        blank_ld        # blanking counter load
        scan_ld;        # scan counter load

rs=0 outputs
        blank_dec       # blanking counter enable
        scan_dec        # scan counter enable
        sync_out;       # scan output--hscan for LSB chip,
                        #   vscan for MSB chip

state idle_state
        sync_en & !sel --> hstart_state scan_ld=1 scan_dec=1
                blank_ld=1 blank_dec=1,
        sync_en & sel & !sync_in --> vscan_h0_state
                scan_ld=1 scan_dec=0,
        sync_en & sel & sync_in & scan_zero --> vblank_h1_state
                scan_ld=1 blank_ld=1 blank_dec=1,
        sync_en & sel & sync_in & !scan_zero --> vscan_h1_state
                scan_ld=1 blank_ld=1 scan_dec=1 sync_out=1,
                --> idle_state sync_out=0;

state hstart_state
        scan_zero --> hblank_state scan_ld=1 sync_out=0,
        --> hscan_state blank_ld=1 sync_out=1;

state hscan_state
        !sync_en --> idle_state,
        scan_zero --> hblank_state scan_ld=1 sync_out=0,
                --> hscan_state blank_ld=1;

state hblank_state
        !sync_en --> idle_state,
        blank_zero --> hscan_state blank_ld=1 sync_out=1,
                --> hblank_state scan_ld=1;

state vscan_h0_state
        !sync_en --> idle_state,
        sync_in --> vscan_h1_state
```

```
                     blank_ld=1 scan_dec=0 sync_out=1,
                --> vscan_h0_state blank_ld=1
                        scan_dec=0;

state vscan_h1_state
        !sync_en --> idle_state,
        !sync_in & scan_zero --> vblank_h0_state blank_dec=0
                                sync_out=0 scan_dec=0,
        !sync_in & !scan_zero --> vscan_h0_state blank_ld=1
                                scan_dec=1 sync_out=1,
                --> vscan_h1_state blank_ld=1 scan_dec=0;

state vblank_h0_state
        !sync_en --> idle_state,
        sync_in & blank_zero --> vscan_h1_state scan_dec=1
                                sync_out=1,
        sync_in & !blank_zero --> vblank_h1_state scan_ld=1
                                blank_dec=1,
                --> vblank_h0_state scan_ld=1 blank_dec=0;

state vblank_h1_state
        !sync_en --> idle_state,
        !sync_in --> vblank_h0_state
                     scan_ld=1 blank_dec=0 sync_out=0,
                --> vblank_h1_state scan_ld=1 blank_dec=0;

end


# 5/29/91 (faust)
#   - added state outputs for test purposes.

# 2/27/92 (faust)
#   - changed states vscan_h1 and vscan_h0 so that vertical
#     scan counts on FALLING edge of hsync; vertical blank
#     still counts on RISING edge of hsync.

# 3/3/92 (faust)
#   - fixed bug resulting from 2/27/92 change--vscan_h1_state
#     was going to vblank_h1_state instead of vblank_h0_state
#     on falling edge of hsync.
```

[la]sync_gen by nhm on 16-Feb-93 at 12:22 A.M.

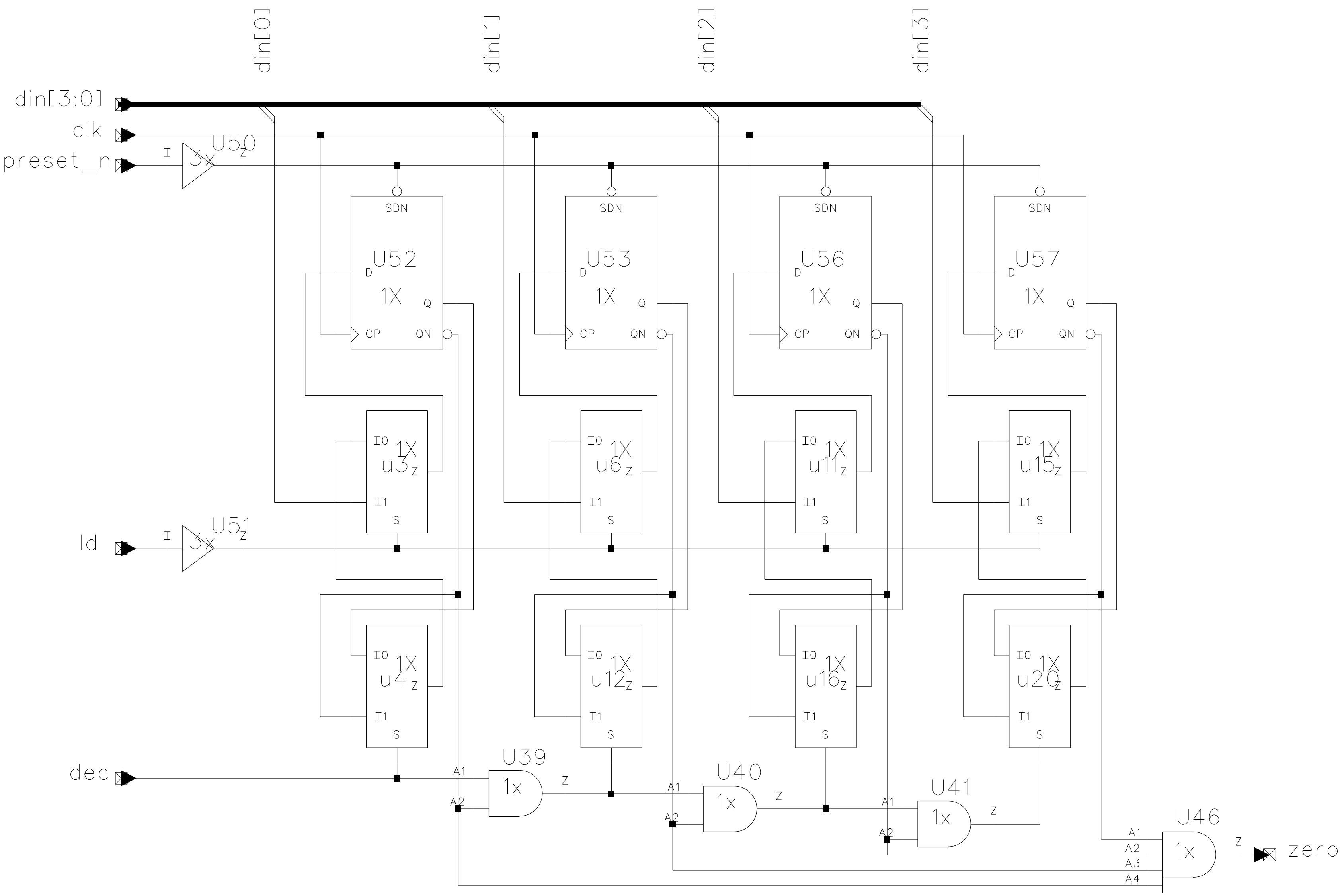


Controller chips

sync_gen.ps

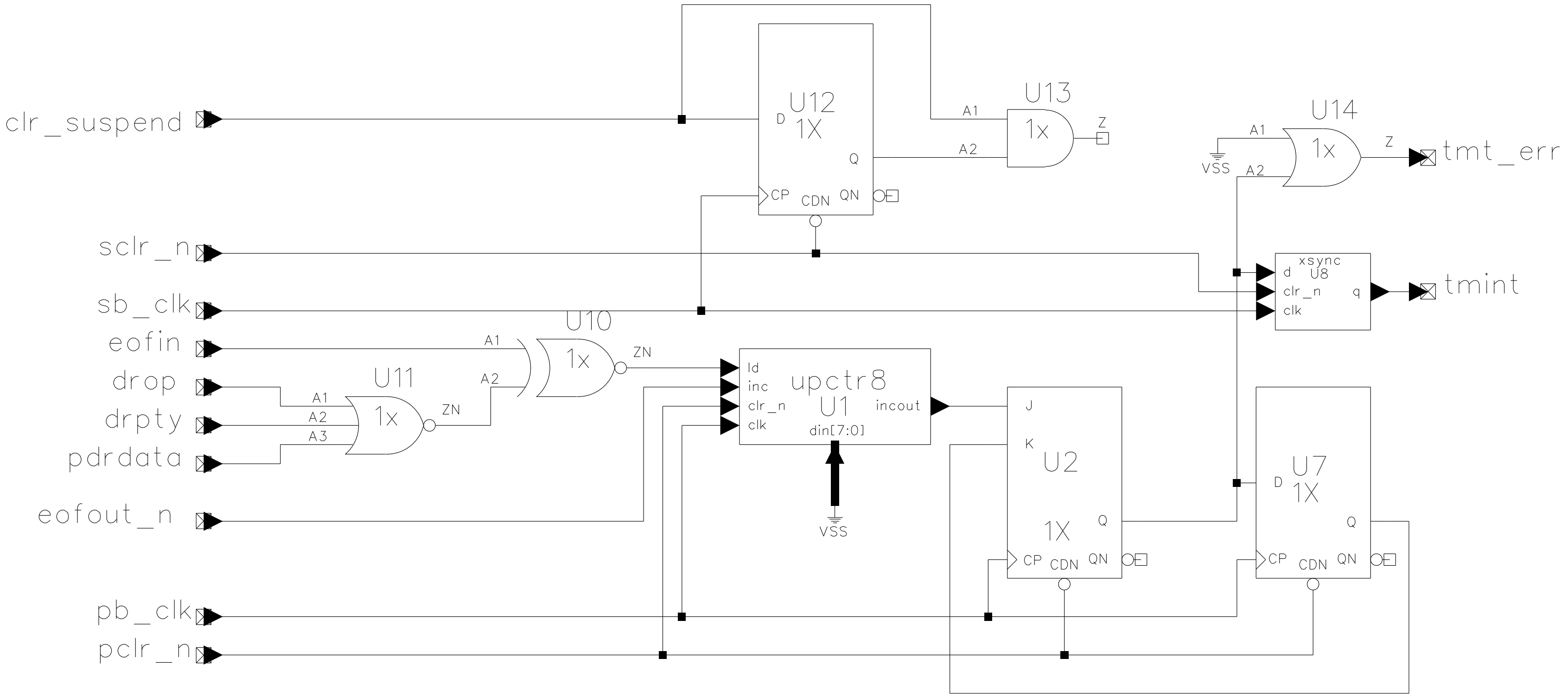
[la]timer4 by nhm on 16-Feb-93 at 12:22 A.M.

timer4.ps



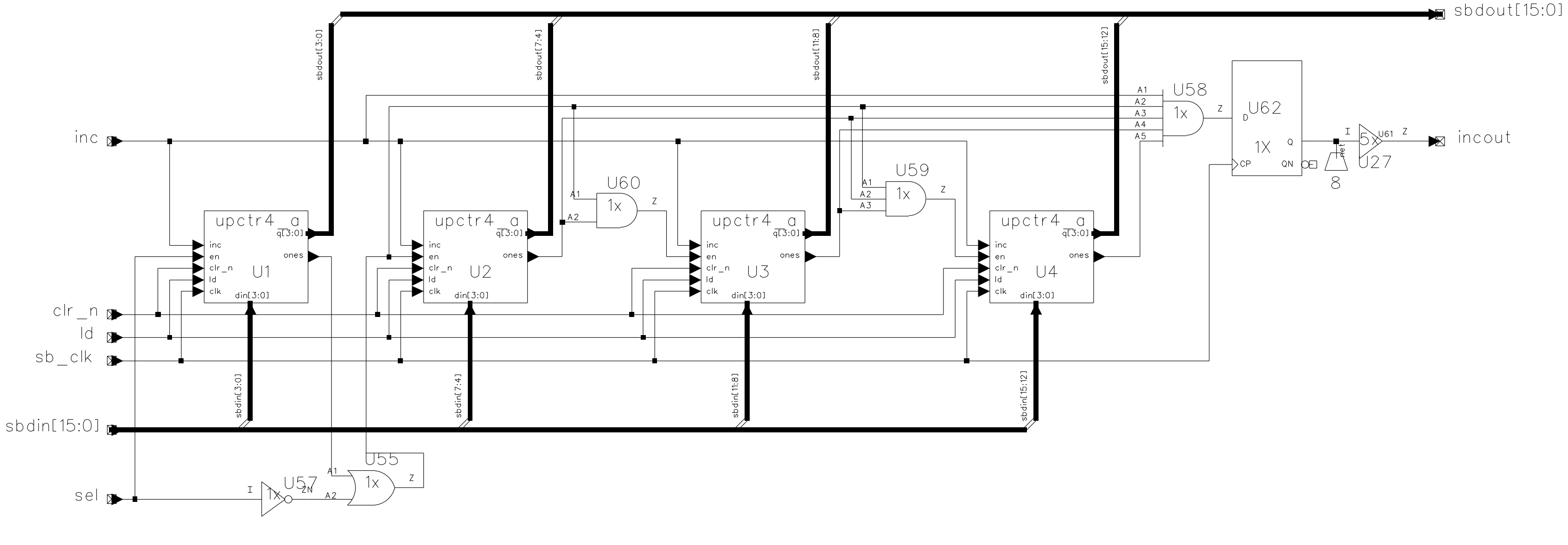
Controller chips

Note: A two clock wide clr_suspend pulse
generates a timeout for test purposes.
This cannot occur in normal operation.
Note: This scheme is no good because u13.z glitches!



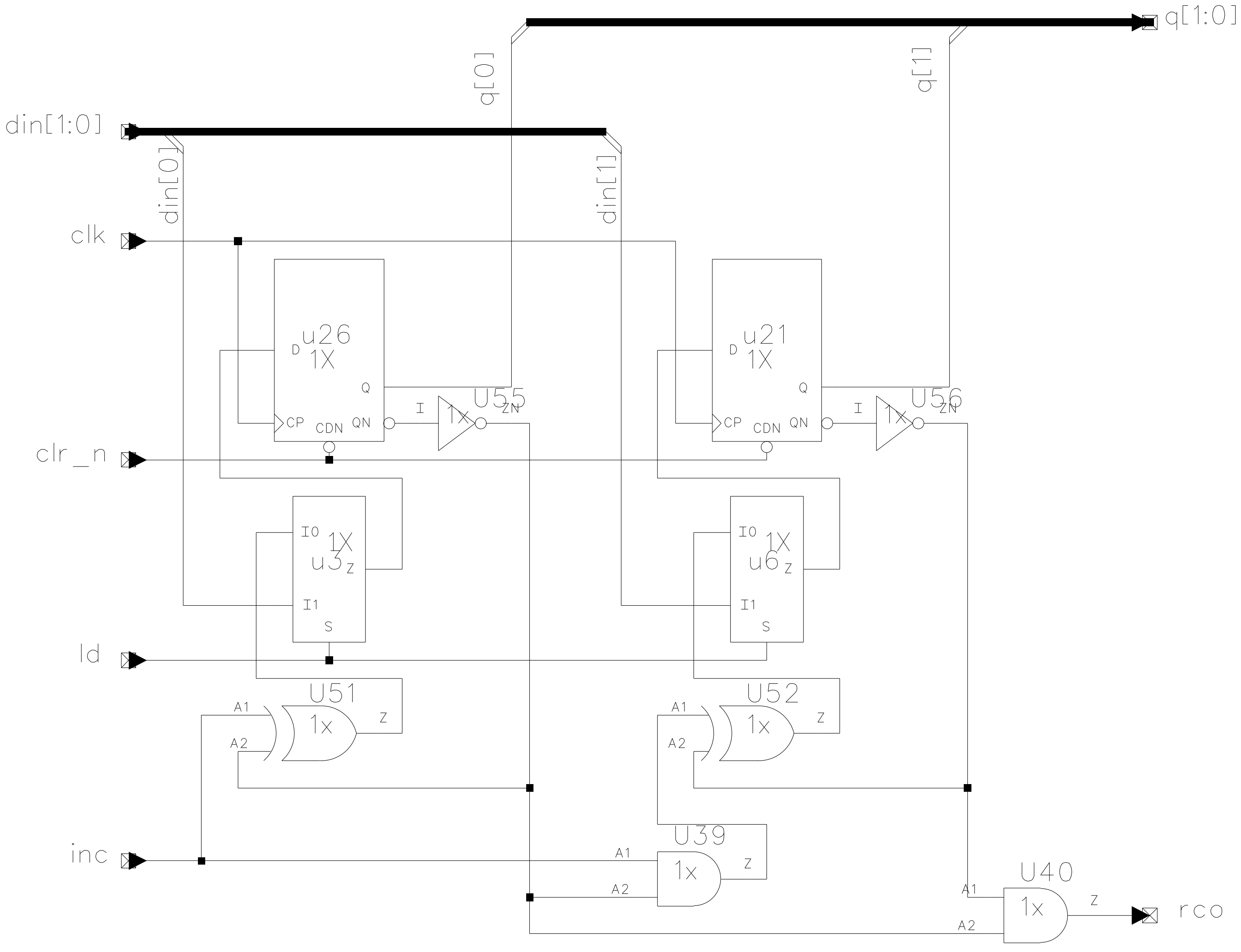
Controller chips

tmctr.ps



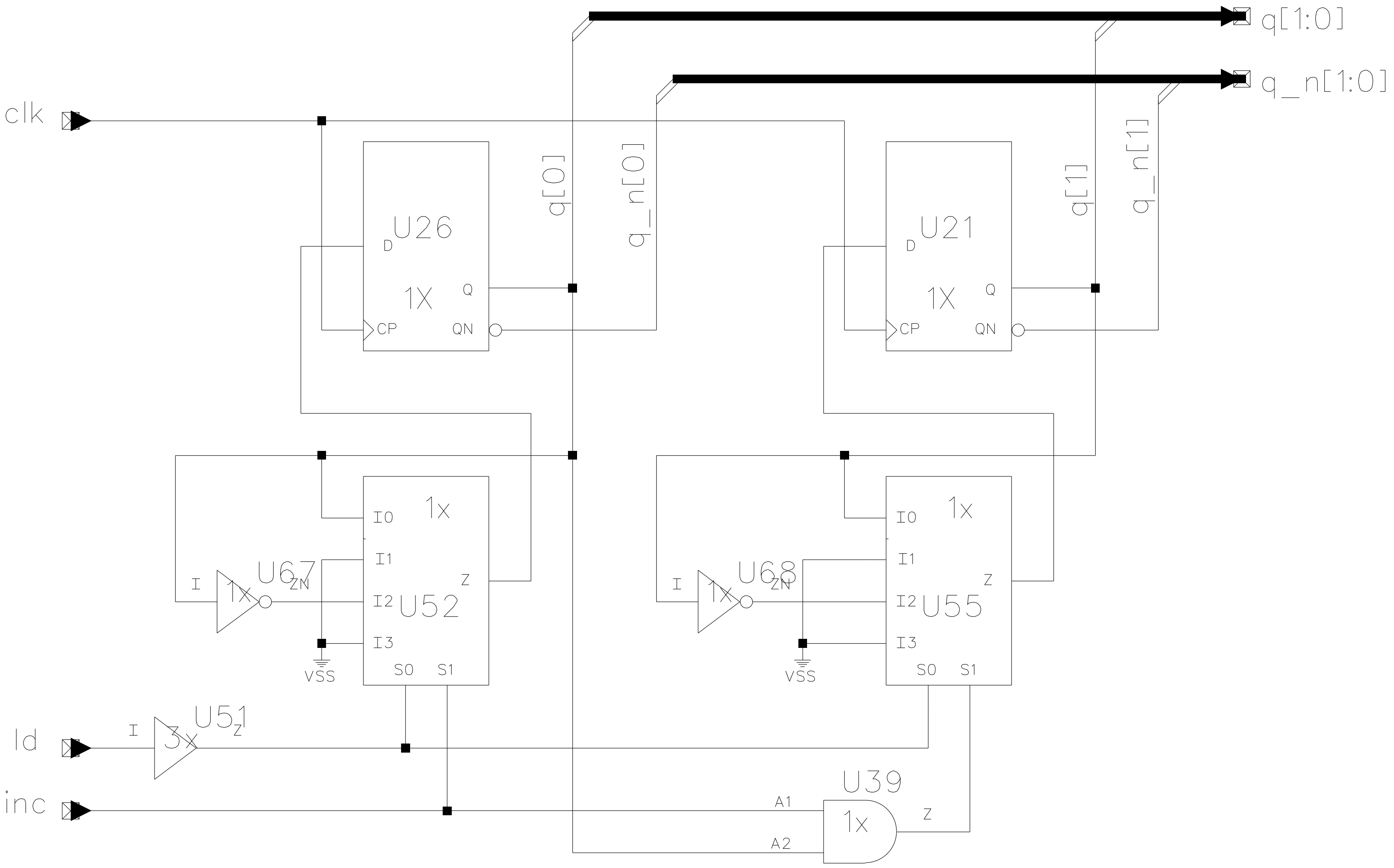
Controller chips

[la]upctr2 by nhm on 16-Feb-93 at 12:22 A.M.

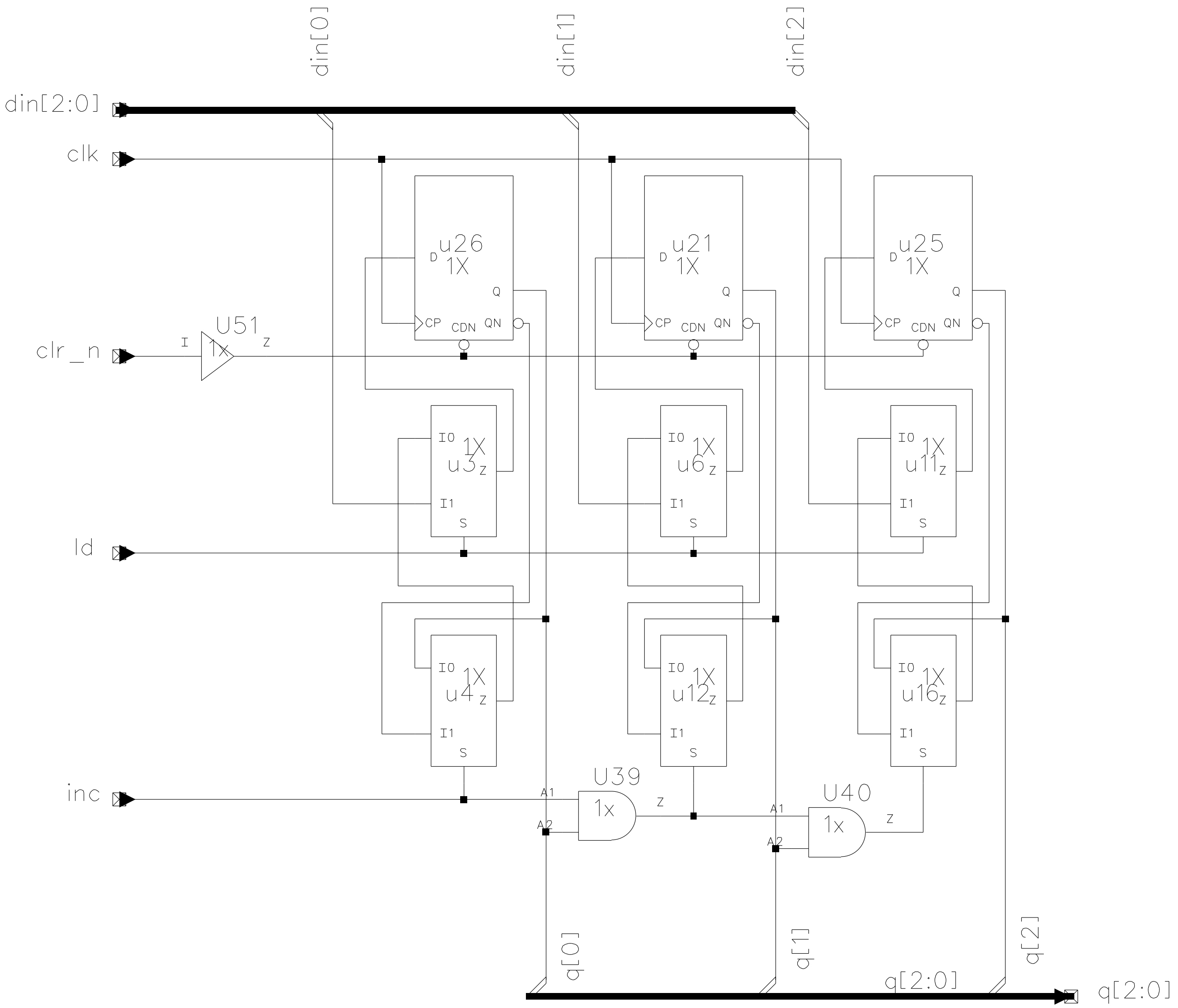


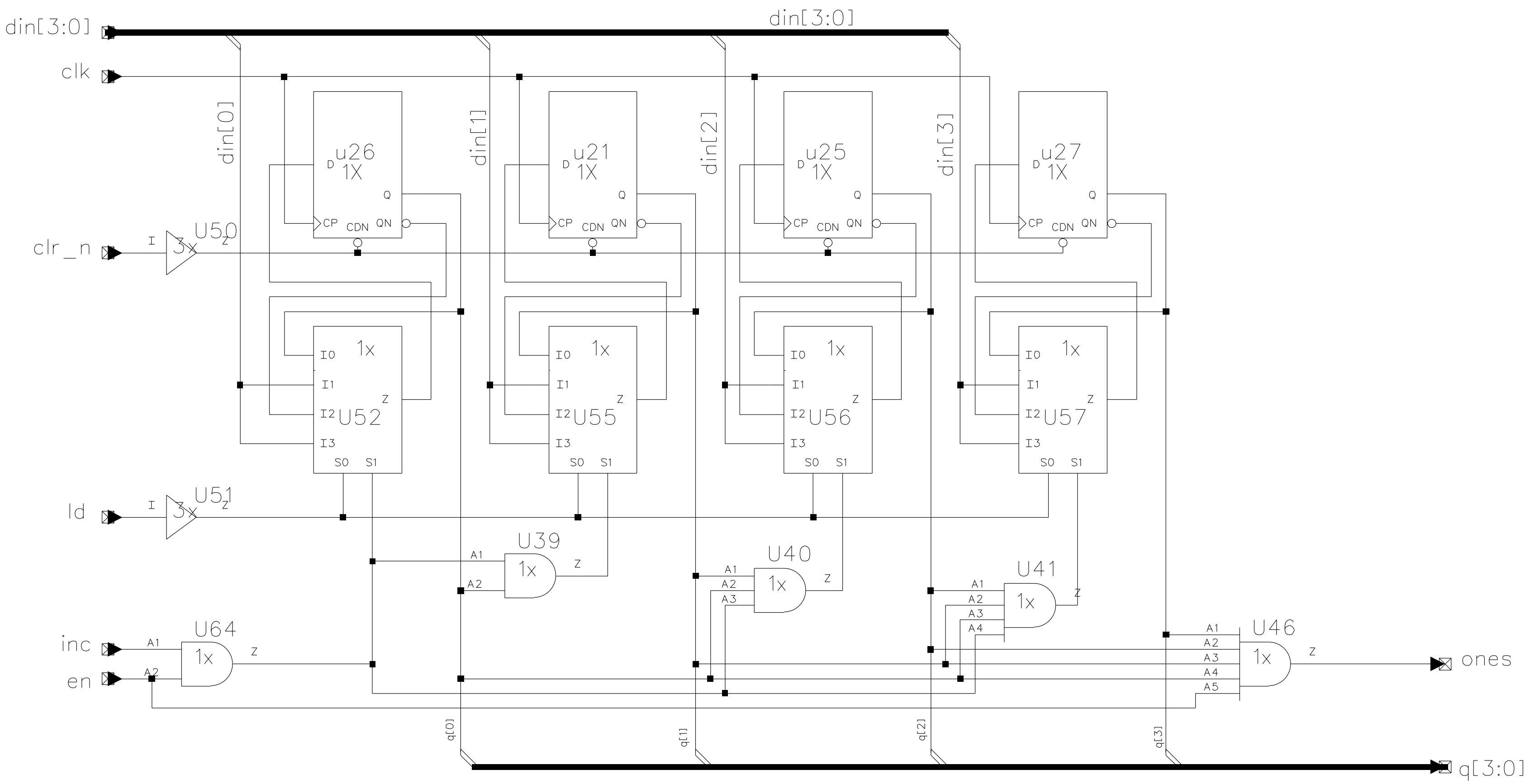
Controller chips

upctr2.ps



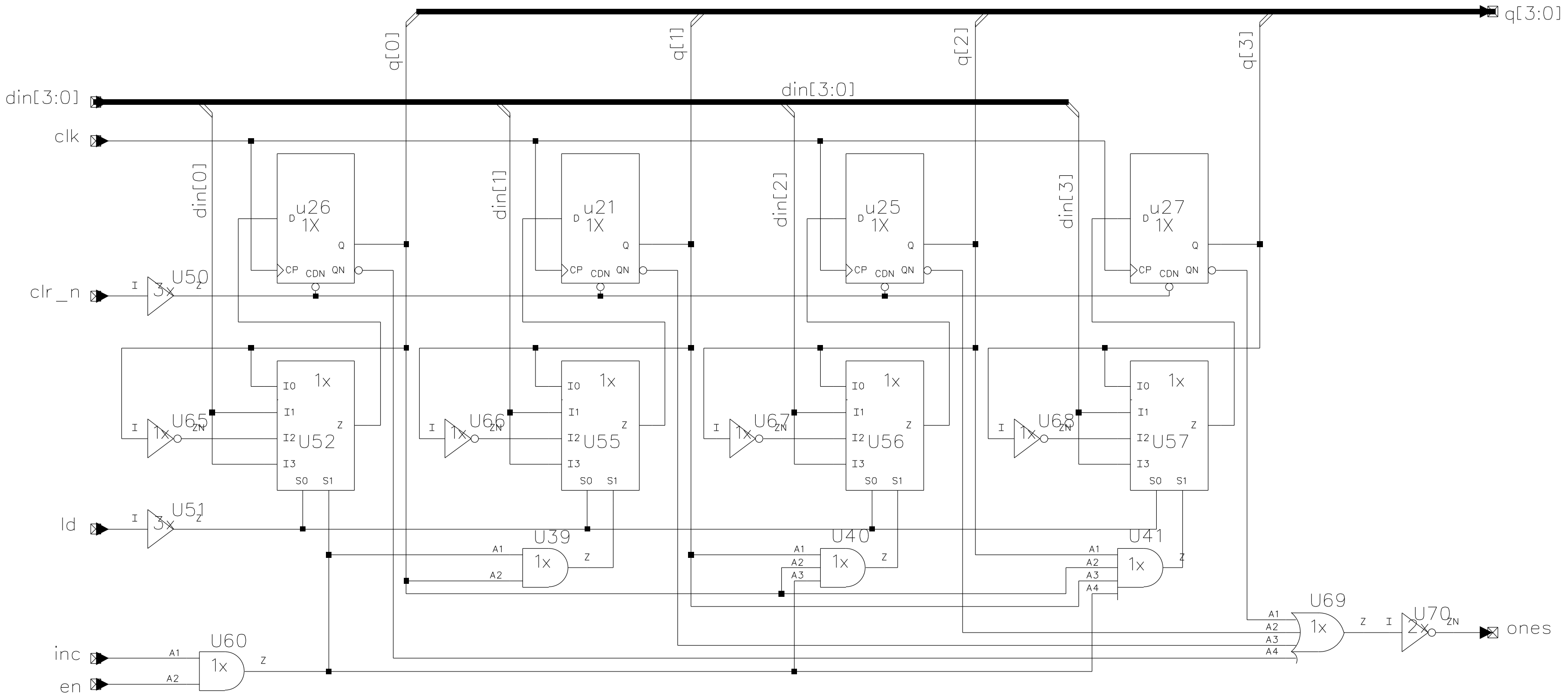
Controller chips


din[0]
din[1]
din[2]
din[2:0]
clk
clr_n
ld
inc
u26
u21
u25
1X
D
Q
CP
CDN
QN
U51
I
Z
1x
I0
I1
S
Z
u3
u6
u11
u4
u12
u16
U39
U40
A1
A2
q[0]
q[1]
q[2]
q[2:0]



Controller chips

[la]upctr4_a by nhm on 16-Feb-93 at 12:22 A.M.

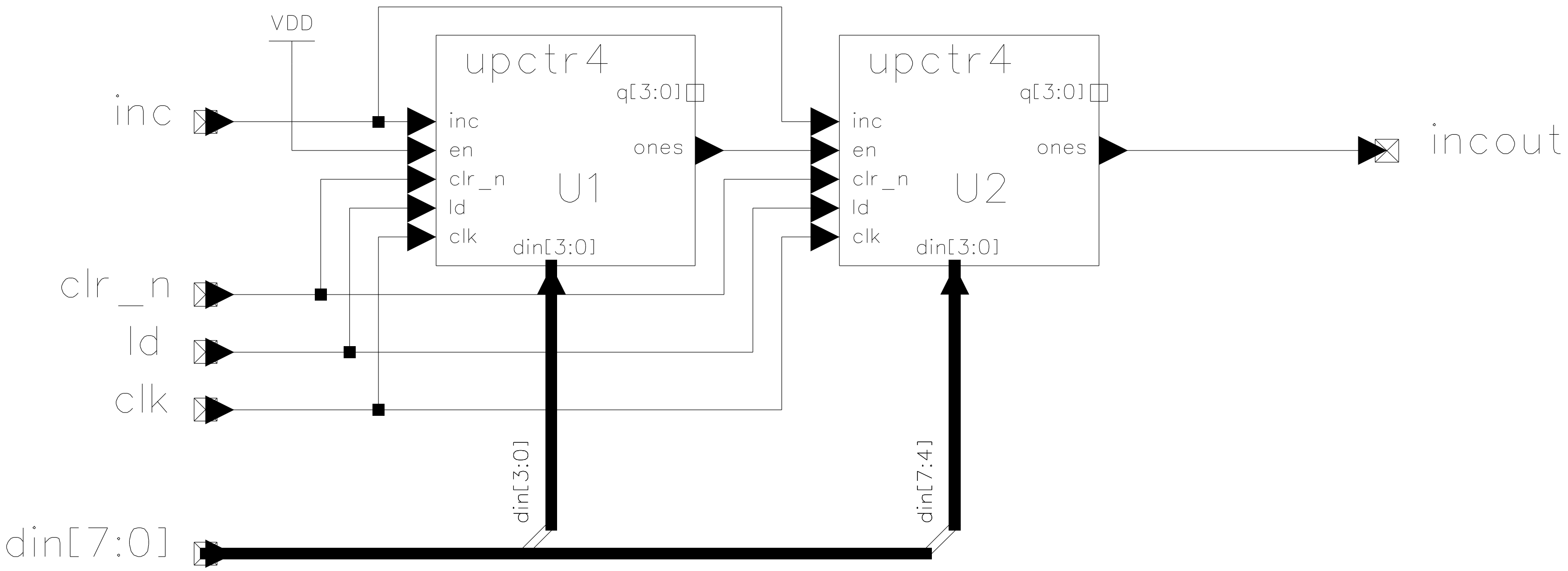


Controller chips

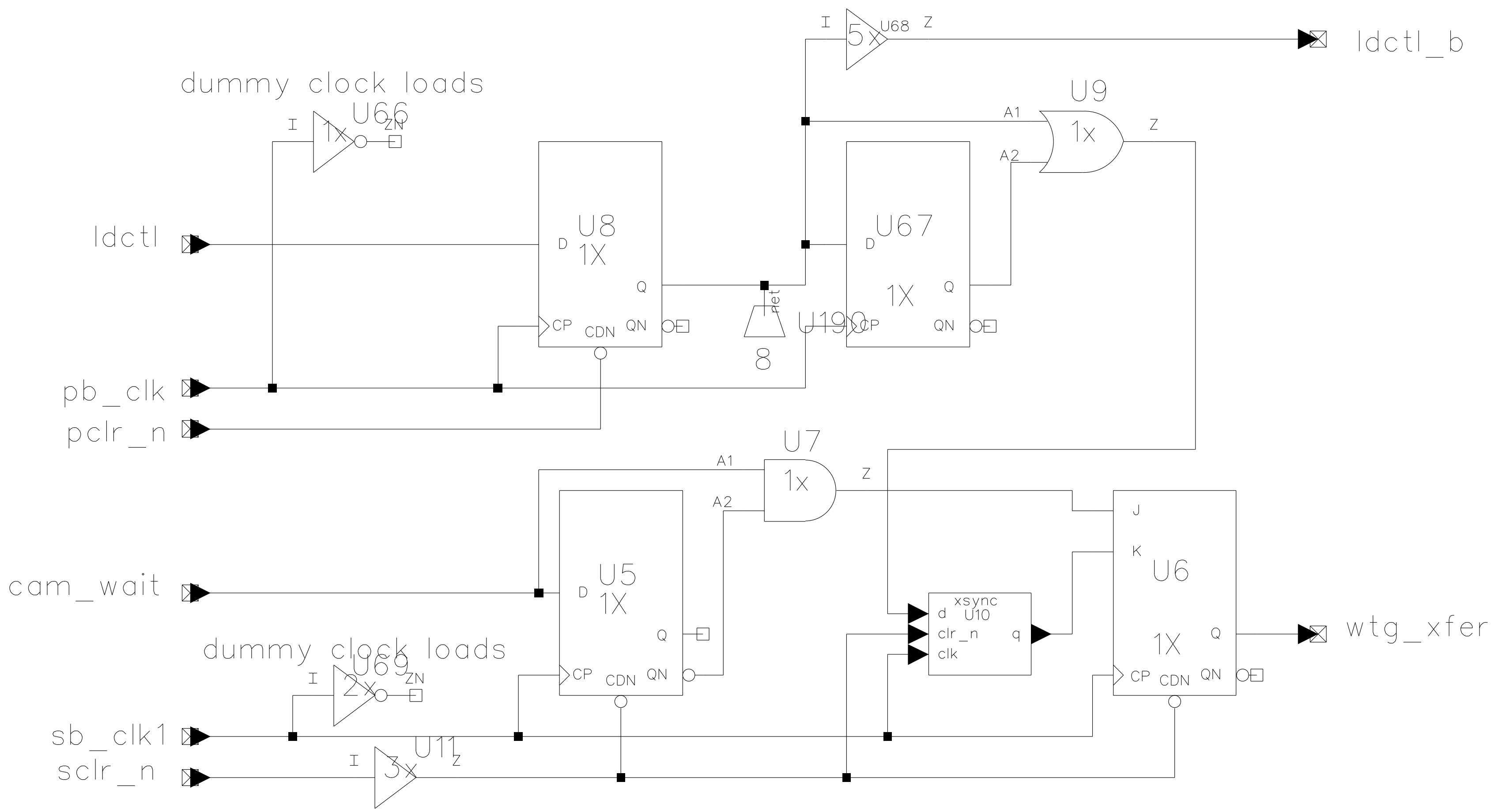
upctr4_a.ps

[la]upctr8 by nhm on 16-Feb-93 at 12:22 A.M.

upctr8.ps



Controller chips

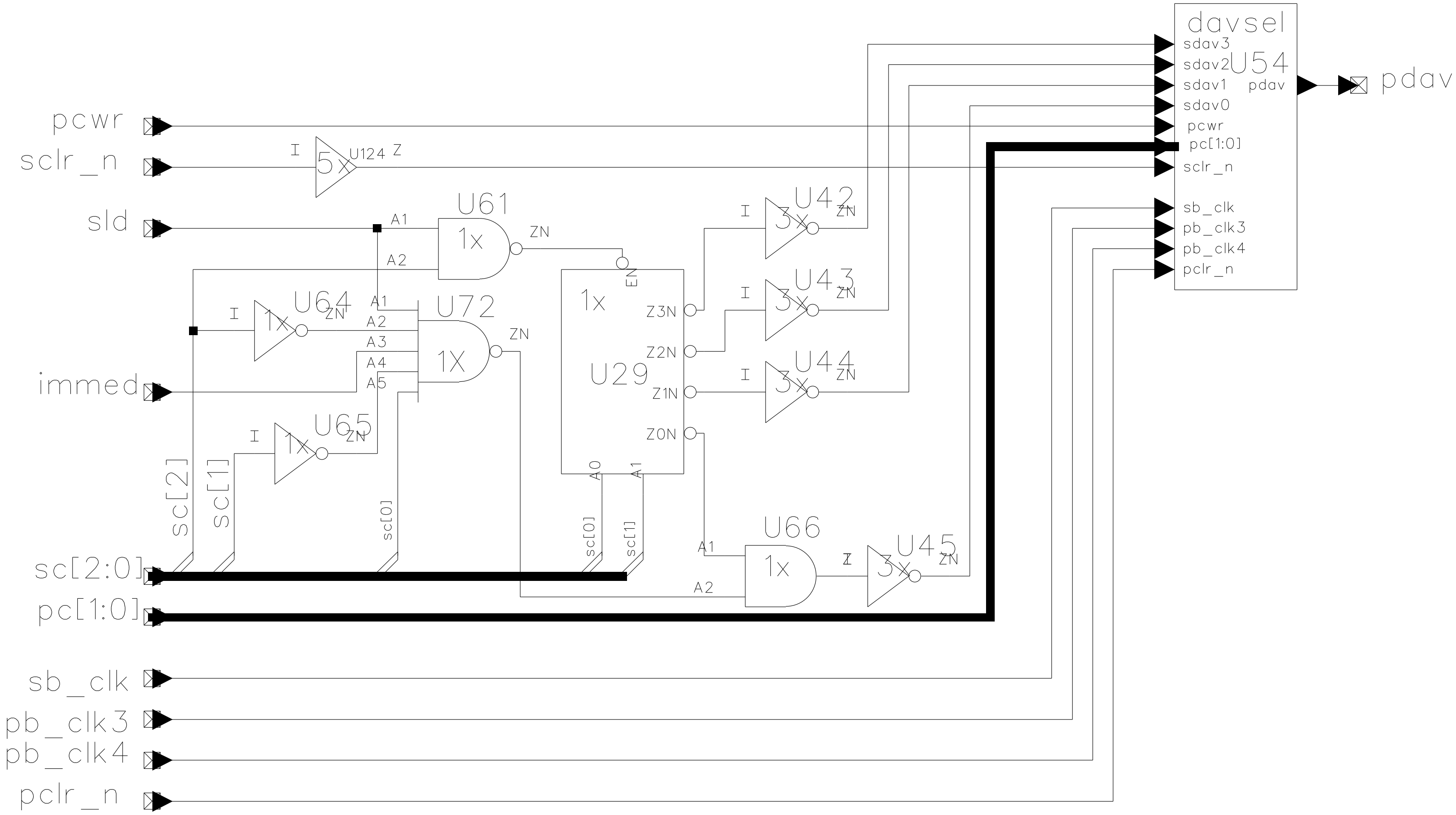


Controller chips

wait_stat.ps

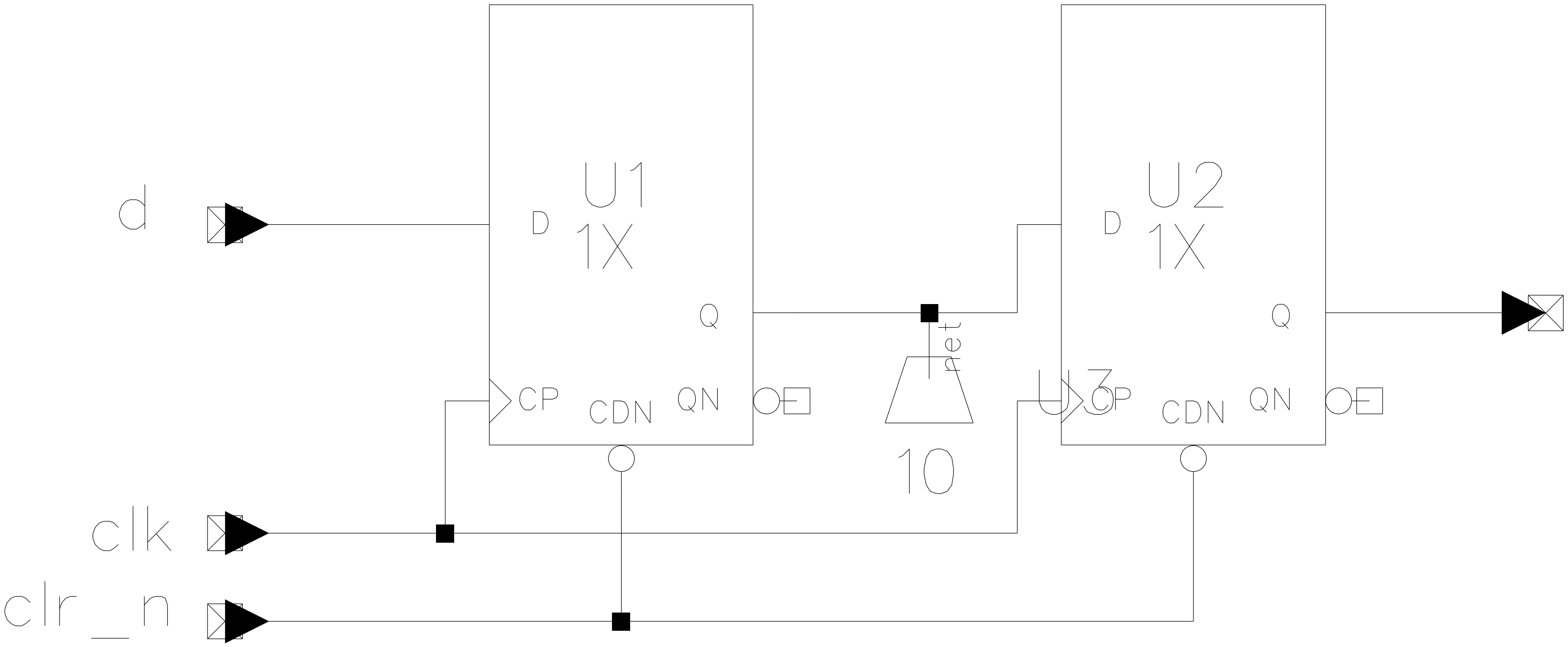
[la]wr_dav by nhm on 16-Feb-93 at 12:23 A.M.

wr_dav.ps



Controller chips

[la]xsync by nhm on 16-Feb-93 at 12:23 A.M.



Controller chips

xsync.ps