Slide Set 1 | Noise

Damage	0	?	1	Damage

Vmin V0 VIL VIH V1 Vmax

-0.3V 0.7V 2.5V

0V 1.7V 2.8V

Combinational Logic
- only depends on present inputs
- No memory

Sequential Logic
- Has memory
- includes feedback
- Synchronous uses a clock

Boolean Properties

Communicative	$X \wedge Y = Y \wedge X$	$X \vee Y = Y \vee X$
Associative	$X \wedge (Y \wedge Z) = (X \wedge Y) \wedge Z$	$X \vee (Y \vee Z) = (X \vee Y) \vee Z$
Distributive	$X \wedge (Y \vee Z) = (X \wedge Y) \vee (X \wedge Z)$	$X \vee (Y \wedge Z) = (X \vee Y) \wedge (X \vee Z)$
Idempotence	$X \wedge X = X$	$X \vee X = X$
Absorption	$X \wedge (X \vee Y) = X$	$X \vee (X \wedge Y) = X$
Combining	$(X \wedge Y) \vee (X \wedge \bar{Y}) = X$	$(X \vee Y) \wedge (X \vee \bar{Y}) = X$
DeMorgan's	$\overline{(X \wedge Y)} = \bar{X} \vee \bar{Y}$	$\overline{(X \vee Y)} = \bar{X} \wedge \bar{Y}$

Axioms

	$0 \wedge X = 0$	$1 \wedge X = X$
	$1 \vee X = 1$	$0 \vee X = X$
Identity		
Idempotence		
	$\bar{\bar{X}} = X$	$T = 0$
negation		

Sum of Products Normal Form

row in truth table with output = 1

$f(a,b,c) = (\bar{a} \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c}) \ldots\ldots$

Dual Functions

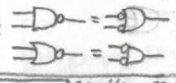
$f^D(\bar{a}, \bar{b}, \ldots\ldots) = \overline{f(a,b,\ldots\ldots)}$

Switch:

$\wedge - \vee$	0-1
$\vee - \wedge$	1-0

*Do not change Complements

DeMorgan's Law



Slide Set 2

Boolean → Verilog

& AND
| OR
~ NOT
^ XOR

Cool Things

2'b arrayx: //y = 8'b10101010101

reg [3:0] data [7:0]; //data contains 8 elements each with 4 bits

Bitwise expression:

wire [3:0] result = A & B;
//result[0] = A[0] & B[0]..... Bitwise of one element:

wire result = |A;
//result = (A[0] | A[1] | A[2].....)

Verilog literal numbers

<size>'[<signed>]<radix><value>

ex: 8'sb0001_1101

-5, it ommited unsigned

Replicator {k{n}}

wire [1:0] x = 2'b10;
wire [7:0] y = {4{x}};

Binary	one-hot
000	00001
001	00010
010	00100
⋮	⋮

Named Port Association *order doesn't matter, connect directly

Mux3 #(16) m1(a2, a1, a0, s, b);
Mux3 #(16) m1(.a(a0), .b(a1), .c(a2), .sel(s), .out(b));

Test Bench 1

```
module test_maj;
  reg [2:0] count;
  wire out;
  Majority m(count[0], count[1], count[2], out);
  initial begin
    count = 3'b000;
    repeat(8) begin
      #100;
      $display("in = %b, out = %b", count, out);
      count = count + 3'b001;
    end
  end
endmodule
```

Test Bench 2

```
module q7_tb;
  reg [2:0] A;
  reg B, C, D, E;
  wire F;
  q7 DUT(A,B,C,D,E,F);
  initial forever begin
    E=0; #5;
    E=1; #5;
  end
  initial begin
    A=3'b000; B=0; C=0; D=1; #10;
    A=3'b111; D=0; #10;
    C=1; B=1; #10;
    D=1; #10;
    $stop;
  end
endmodule
```

Always Block *anything assigned must be a reg

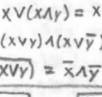
```
always @(sel or a or b) begin
  // order matters
end
```

Rules:
1. Every input is in sensitivity list or @(*)
2. Must have an output for every combo of inputs or default

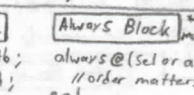
Slide Set 3

Karnaugh Maps

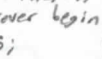

Two-Variable


Three-Variable

four-Variable

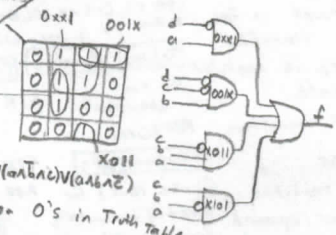

five-Variable

Hazards



Definitions

minterm → row in the truth table
implicant → row with output = 1
PI → all circles
EPI → circle with exposed 1 (only covered once)



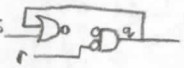
d	c	b	a	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

$f = (a \wedge \bar{J}) \vee (b \wedge \bar{c} \wedge \bar{J}) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c})$

Product of Sums → Focus on 0's in Truth Table

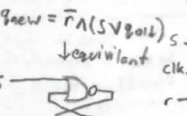
Slide Set 4/5

Reset-Set (RS) Latch



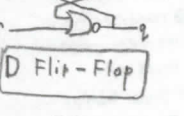
r=1 → q=0
r=0, s=1 → q=1
r=0, s=0 → q holds last value

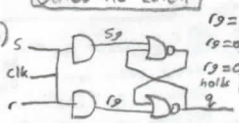
Gated RS Latch



$q_{new} = \bar{r} \wedge (s \vee q_{old})$

↓ equivalent

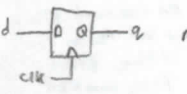




rg=1 → q=0
rg=0, sg=1 → q=1
rg=0, sg=0 → q holds last value

*when clk is high, acts as a RS latch
*when clk is low, maintains value regardless of s, r
ie: level sensitive

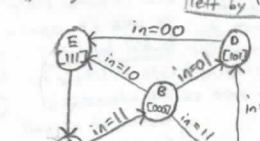
D Flip-Flop



q=d on clk rising edge

n-bit register?
- Flip flop for each bit all connected to same clk

```
module FSM(clk, reset, in, out);
  input clk, reset;
  input [1:0] in;
  output [2:0] out;
  `define Sa 3'b000
  `define Sb 3'b001
  `define Sc 3'b010
  `define Sd 3'b011
  `define Se 3'b111
  wire [2:0] present_state, state_next_reset, state_next;
  reg [6:0] next;
  VDFF #(3) STATE(clk, state_next_reset, present_State);
  assign state_next_reset = reset ? `Sa : state_next
  always @(*) begin
    casex({in, present_state})
      {2'b11, `Sa} : next = {`Sb, 3'b101};
      {2'bxx, `Sa} : next = {`Sa, 3'b101};
      {2'b10, `Sb} : next = {`Se, 3'b000};
      {2'b01, `Sb} : next = {`Sd, 3'b000};
      {2'b11, `Sb} : next = {`Se, 3'b000};
      {2'bxx, `Sb} : next = {`Sb, 3'b000};
      ⋮
      default: next = {`Sa, 3'b101};
    endcase
  end
  assign {state_next, out} = next;
endmodule
```



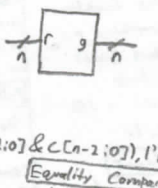
D Latch

clk=high → q = D
clk=low → q maintain

Non Blocking Assignment
- Signals are updated at the same time at the end of the always block (<=)

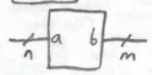
Arbiter
- Finds a 1 starting at LSB and outputs a one hot-code as the position of that one

```
module Arb(r,g);
  parameter n=8;
  input [n-1:0] r;
  output [n-1:0] g;
  wire [n-1:0] c;
  wire [n-1:0] g;
  assign c = {(r<<[n-2:0] & c[n-2:0]), 1'b1};
  assign g = r & c;
endmodule
```




```
module Dec(a,b);
  parameter n=2;
  parameter m=4;
  input [n-1:0] a;
  output [m-1:0] b;
  wire [m-1:0] b = 1<<a;
endmodule
```

Decoder



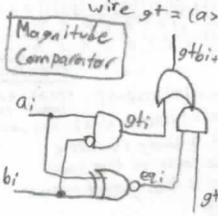
Binary → 1-Hot

Equality Comparator

```
module EqComp(a,b,eq);
  parameter k=8;
  input [k-1:0] a, b;
  output eq;
  wire eq;
  assign eq = (a==b);
endmodule
```

```
module MagComp(a,b,gt);
  parameter k=8;
  input [k-1:0] a,b;
  output gt;
  wire gt = (a>b);
```

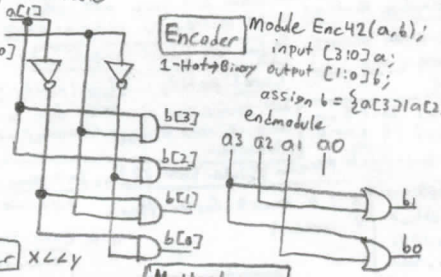
Magnitude Comparator



Encoder

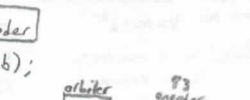
1-Hot → Binary

```
module Enc42(a,b);
  input [3:0] a;
  output [1:0] b;
  assign b = {a[3]|a[2], a[3]|a[1]};
endmodule
```



Priority Encoder

```
module PE(r,b);
  input [7:0] r;
  output [2:0] b;
  wire [7:0] g;
  Arb #(8) a(r,g); • Finds position of first 1 in a and outputs binary of that position
  Enc83 e(g,b);
endmodule
```



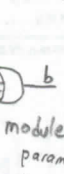
Shifter

x << y
- Shift x to the left by y bit positions

Multiplexer

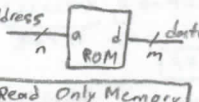
4-input k-wide mux

```
module Mux4(a3,a2,a1,a0,s,b);
  parameter k=1;
  input [k-1:0] a0,a1,a2,a3;
  input [3:0] s;
  output [k-1:0] b;
  reg [k-1:0] b;
  always @(*) begin
    case(s)
      4'b0001 : b = a0
      4'b1000 : b = a3
      default: b = {k{1'bx}};
    endcase
  end
endmodule
```



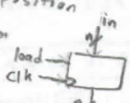
Read Only Memory

```
module ROM(read_address, dout);
  parameter data_width = 32;
  parameter addr_width = 4;
  parameter filename = "data.txt";
  input [addr_width-1:0] read_address;
  output [data_width-1:0] dout;
  reg [data_width-1:0] mem [2**addr_width-1:0];
  initial $readmemb(filename, mem);
  assign dout = mem[read_address];
endmodule
```




```
module VDFFE(clk, en, in, out);
  parameter n=1;
  input clk, en;
  input [n-1:0] in;
  output [n-1:0] out;
  reg [n-1:0] out;
  wire [n-1:0] next_out;
  assign next_out = en ? in : out;
  always @(posedge clk)
    out = next_out;
endmodule
```

Register with load enable



2's Complement

Steps:
1. Flip all bits
2. Add one

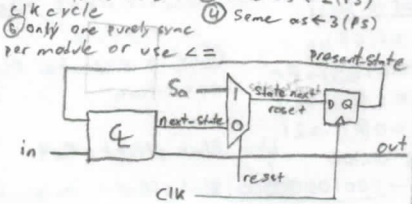
-8	4	2	1	
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
⋮				
0	0	0	0	
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3

Purely Synchronous
1. only clk in sensitivity list
2. only signals that change on the same edge in the same always block
3. clk should not be read/written inside
4. Block can only describe what happens in 1 clk cycle
5. only one purely sync per module or use <=

Purely Level Sensitive (will give inferred latch)

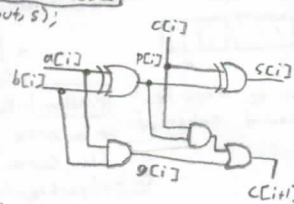
```
always @(*)
  if (clk == 1'b1)
    z = A && B;
```

1. use only (*) for sensitivity list
2. clk only in 1 if condition
3. Same as ←2 (PS)
4. Same as ←3 (PS)



Multi-Bit adder

```
module Adder1(a,b,cin,cout,s);
  parameter n=8;
  input [n-1:0] a,b;
  input cin;
  output [n-1:0] s;
  output cout;
  wire [n-1:0] s;
  wire cout;
  assign {cout,s} = a+b+cin;
endmodule
```



Pipeline Cycles

	W	W	W
LDR r1, [r2, #4]	M	M	M
STR r1, [r3, #8]	X	X	X
ADD r1, r1, r1	D	S	S
	F	D	D
		F	F

parameter data_width = 32;
parameter addr_width = 4;
parameter filename = "data.txt"
input clk;
input [addr_width-1:0] read_address, write_address;
input write;
input [data_width-1:0] din;
output [data_width-1:0] dout;
reg [data_width-1:0] dout;
reg [data_width-1:0] mem [2**addr_width-1:0];
initial $readmemb(filename, mem);
always @(posedge clk) begin
if(write)
mem[write_address] <= din;
dout <= mem[read_address];
end
endmodule

Note:
STMFD r13!, {r0-r7, r14}
Push {R0-R7, lr}

module Add_Sub(a, b, sub, s, ovf);
parameter n=8;
input [n-1:0] a, b;
input sub; //subtract if sub=1 else add
output [n-1:0] s;
output ovf;
wire c1, c2;
wire ovf = c1 ^ c2;
Adder1 #(n-1) a1(a[n-2:0], b[n-2:0]^{n-1{sub}}, sub, c1, s[n-2:0]);
Adder1 #(1) a2(a[n-1], b[n-1]^sub, c1, c2, s[n-1]);
endmodule

Instruction Encoding
Slide Set 9
COND → usually E
F → instruction format 0 for processing, 1 for transfer
I → if 0, Operand2 is register otherwise immediate
Opcode → specifies operation
S → for branches
Rd → Destination register
Rn → first register operand
Operand2 → second operand

Example Encodings
SUB sp,sp,#41 → E24DD004
LDR r8,[sp] → E59D8000
STR r10,[sp,#28] → E58DA01C
CMP r2,r3 → E1520003
BLE → DA000001
MVN r0,#0 → E3E00000
MOV PC,LR → E1A0F00E
ADD r6,r2,r5,LRS #1 → E0826045
ADD r7,r0,r6,LSL #2 → E0807106
ASR r9,r8,#0 → E26B9000

Logical operations
AND r5,r1,r2 //r5 = r1 & r2
//bitwise AND/OR
ORR r5,r1,r2 //r5 = r1 | r2
MVN r5,r2 //r5 = ~r2
ADD r3,r1,r3,LSL #2 //r5 = r1 + (r3<<2)
MOV r6,r5,LSR r3 //r6 = r5 >> r3
LDR r5,[r3,r4,LSL #2] //r5 = A[i]
//where r3 is base address, r4 is index

Indirect Branching
Bx rm
↓equivilant
MOV PC,rm

Function Calls
① use r0-r3 for first 4 parameters, save the rest on stack
② BL to the function
③ Save everything on stack (save down)
④ Perform task
⑤ Put result in r0
⑥ mov pc,lr

Conventions
•not expected to save r0-r3, r12
•Expected to save r4-r11

Post Indexed addressing
STR r3,[r2],#4
↓equivilant
STR r3,[r2]
ADD r2,r2,#4

C-Code Example
while(save[i]==k)
i+=1;

Loop: ADD r12,r6,r3,LSL#2
LDR r0,[r12,#0]
CMP r0,r5
BNE Exit
ADD r3,r3,#1
B Loop
Exit:

Branch Target Encoding
PC_Target = PC_Branch + 8 + 4*"Target address"

Spill/Fill
SUB sp,sp,#16
STR r6,[sp,#12]
STR r5,[sp,#8]
STR r4,[sp,#4]
STR lr,[sp,#0]
PUSH {r4-r6,LR}
LDR lr,[sp,#0]
LDR r4,[sp,#4]
LDR r5,[sp,#8]
LDR r6,[sp,#12]
ADD sp,sp,#16
POP {r4-r6,LR}

Arrays
ARRAY:
•Word 0
•Word 1
⋮
•Word 9
LDR r1,=ARRAY

Slide Set 10 — How? → Problems → Solutions

Interrupt Service Routine
①Recieve interrupt signal at instruction i
②Finish i, then move PC to first instruction of ISR
③Return to "i+1"

Memory Mapped Registers
•Not in Reg File, these are registers associated with I/O device stored in memory

Polling
•Do nothing until a valid I/O interrupt comes
ANDS r4,r4,#1
//AND r4 and 1 and update status flags. Usually only CMP updates status flags

Register Banks
CPSR → Current Program Status register
•Different modes
•Blue is the same exact physical register
•Orange is a different physical register

Exception Handling
•Exception occurs, copies CPSR into SPSR

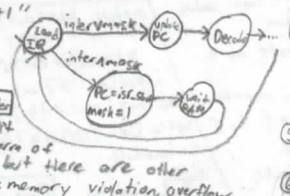
Problems
①Infinite loop, interrupt always asserted in loop
②Return address, how to set back IR state
③Save, what you need when you return to "i+1"
④Stop interrupt, how to tell I/O device we're done
⑤CMP, need to Save Status Flags also
⑥return to "i+1"
⑦Multiple I/O's, more than one interrupt

Solutions
→Add a Mask register we know there's an interrupt now execute
→Can't use lr, so use new register LR_IRQ
→Can't save on stack, so use a separate stack (location in memory)
→read/write a memory mapped register
→Save Status register on interrupt stack
→Set PC to "i+1," restore status flags, unmask, SUBS pc,lr,#4
→Different processor modes, Interrupt controller
→multithreading, threads under 1 CPU running at same time.
Multi-core: threads running on different CPU cores

Ex vs interrupt
•interrupt is one form of exception but there are other types ie: memory violation, overflow

Edge vs Level Sensitive
Level: notify I/O device you are done, ie: read keyboard character
Edge: notify I/O and reset flag in interrupt controller also

Multitasking
•Run A for 20ns, Run B for 20ns,...



Slide Set 11
0 1 1 0 0 1 1 1 = 6.4375
$2^3\ 2^2\ 2^1\ 2^0\ 2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}$

Fixed Point

Error
Absolute error = |V(R(x)) - x|
Relative error = |V(R(x)) - x)/x| / |x|
Where: R(x) is binary 1's and 0's
V(R(x)) is the value of this binary number, x is the actual number

Resolution
•Smallest difference, ie: 2^{-4} for Block (above)
•Absolute error ≤ ½ resolution

accuracy
•max error over input range you can get
Absolute = max(|V(R(x)) - x|)
Relative = max(|V(R(x)) - x)/x|)

Floating Point
$V = m \cdot 2^{e-x}$
Where: m is mantissa
e is exponent
x is bias

Instructions
FLDS → 32 bit FP load
FLDD → 64 bit FP load
FSTS → 32 bit FP load
FADDS → 32 bit Addition
FSUBS → 32 bit Subtract
FMULS → 32 bit multiply
FDIVS → 32 bit Division
last 5 or 0 means double or single precision

Slide Set 12
n = performance(x) / performance(y)
x is n times faster than y (Speedup)
n = execution(y) / execution(x)

Slide Set 13

Temporal locality:
•Use it once likely to use it again

Spatial locality:
•use it likely to use addresses around it as well so take those too

Set Associative:
•Index selects set of blocks as opposed to 1, combo of direct and fully Associative

Write Misses
①write-allocate, if using writeback, read memory into cache
②No-write-allocate, if using write through, just write straight to memory

Cache on Write
①write through, store both cache and memory on every byte, Conservative and slow
②write back, store cache only, fast, must mark block as dirty so you know its not in memory

Iron Law
Execution time = (instruction count)(CPI)(Cycletime)
= (Cycle count)(cycle time)
CPI = Cycle Count / Instruction Count

Pipeline Hazards
①Structural, two or more instructions want the same hardware
Sol'n → second instruction has to wait
Sol'n → replicate the hardware block
②Data, can't send info back in time
Sol'n → forwarding, add wires that get info from further back in the pipeline
③Control, how do you know the next instruction is right
Sol'n → option 1 is Stall, Option 2 is predict branch is not taken If wrong have to correct
IF/ID → ID/Ex → Ex/MEM → MEM/WB

IEEE 754 32 bit FP → 1 (sign) | 8 (127+e) | 23 (mantissa)

263.3 in 32-Bit FP

263	100000111
0.3	01001100111....

SN: $1.00000111 \cdot 2^8$

263.3

sign	exp	mantissa
=0	10000111	00000111001100111...

0.3x2	0.6	0
0.6x2	1.2	1
0.2x2	0.4	0
0.4x2	0.8	0
0.8x2	1.6	1
0.6x2	1.2	1
:	:	0
:	:	0
:	:	1
:	:	1

Slide Set 14
Speedup = Old Execution time / new Execution time

Amdahl's Law
$ExTime_{new} = ExTime_{old}[(1 - fractionEnhanced) + \frac{fractionEnhanced}{SpeedupEnhanced}]$

Can't be enhanced | Could be enhanced

Speedup Max = $\frac{1}{1 - fractionEnhanced}$
Best case possible

Problem –Both threads try to access global variable at the same time.
Sol'n → Synchronization, restrict access until thread 1 is done what it is doing

GPU = Graphics Processing Unit

Memory
DRAM: uses 1 transistor and a Capacitor to store a bit. Cheaper than SRAM, longer then SRAM
SRAM: uses 6 MOS transistors to store a bit, Slow with large systems
Disk Tape: Cheapest
Flash: quantum tunnel in bit

Cache Organizations
Fully Associative: Comparing all tags is expensive in (time, area, energy)
Direct Mapped: Only one tag to be checked because block only stored in single row. Has more conflict misses than fully associative which can use all rows

Block address: Tag | Index | Block offset
Index = (Block address) % # rows
log2(block size) removed from block address

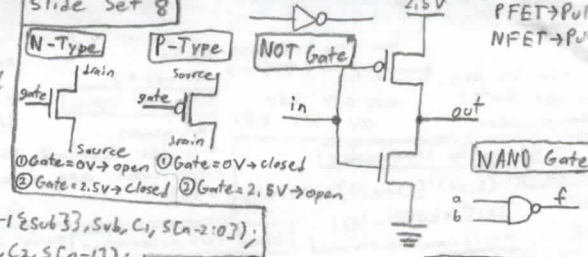
Hex → Decimal
$(AB0\,9)_{16} = (43785)_{10}$
$16^3\ 16^2\ 16^1\ 16^0$
$10 \cdot 16^3 + 11 \cdot 16^2 + 0 + 9 \cdot 16^0 = 43785$

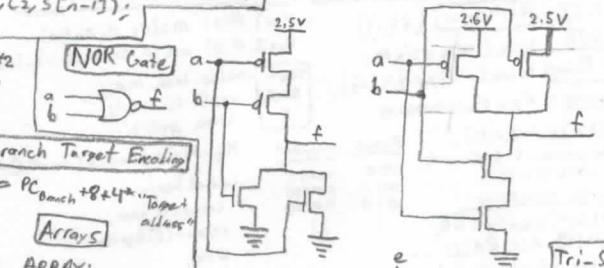
Decimal → Hex
$(479)_{10} = (1DF)_{16}$
479 ÷ 16 = 29 R15 → F
29 ÷ 16 = 1 R13 → D
1 ÷ 16 = 0 R1 → 1
Adress is 0x10C
Binary → 0001000011100 (index | block offset)
tag = 0x10
Block offset = 0x4
Block offset = log2(block size)

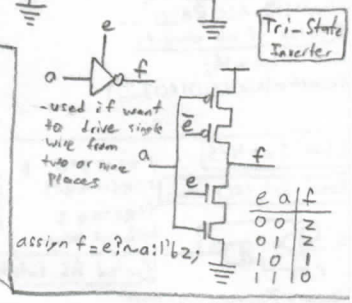
Pipeline Stages
①F → Instruction fetch from memory
②D → instruction decode, read regs in regfile
③X → execute ALU operation
④M → access memory LDR or STR
⑤W → Write result into Reg file

Tag/Index/offset
Two Subcaches
index 0 Search for tag in index 1 each subcache, if not found put in subcache

PFET → Pull-up
NFET → Pull-down







e	a	f
0	0	z
0	1	z
1	0	1
1	1	0