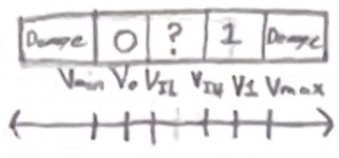
Slide Set 1

Noise



Verilog:

- HDL
- Describes logic gates
- used as input to synthesis/simulation

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

$0 \wedge X = 0$ $\quad 1 \wedge X = X$ $\quad \bar{0} = 1$

$1 \vee X = 1$ (Identity) $\quad 0 \vee X = X$ (Idempotence) $\quad \bar{T} = 0$ (negation)

Sum of Products Normal Form

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

Product term, corresponds to a row in truth table with output = 1

Dual Functions

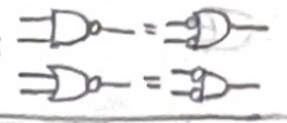
$f^D(\bar{a}, \bar{b}, \ldots) = \overline{f(a,b,\ldots)}$

Switch:

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

* Do not change complements

DeMorgans Law



Slide Set 2

Boolean → Verilog:

& AND
| OR
~ NOT
^ XOR

Cool Things

2Darray:

```
//y = 9'b101010101
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

-s, if omitted unsigned

Replicator

```
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(a1), .b(a0), .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'b100; 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

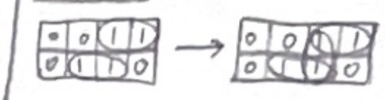
*anything assigned must be a reg

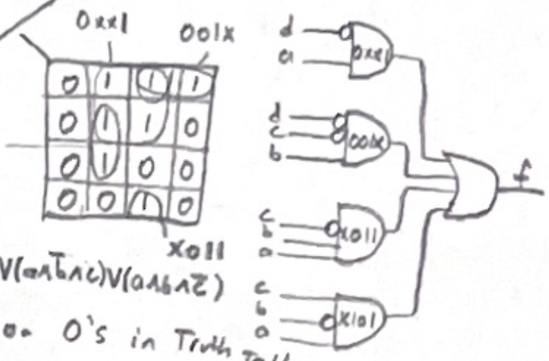
```
always @(sel or a or b) begin
  //order matters
end
```

Rules:

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

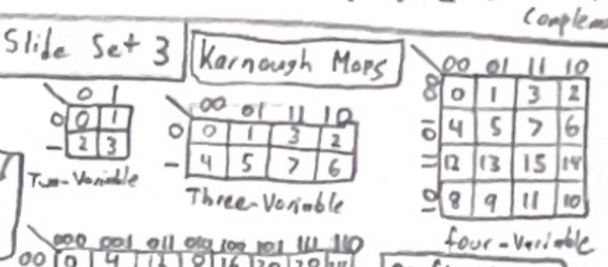
Hazards





Slide Set 3

Karnaugh Maps



Definitions

minterm → row in the truth table

implicant → row with output = 1

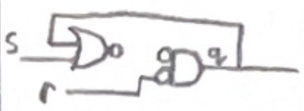
PI → all circles

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	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

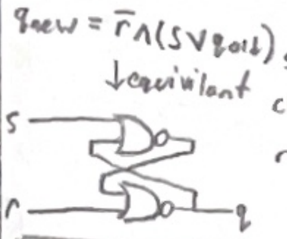
Slide Set 4/5

Reset-Set (RS) Latch

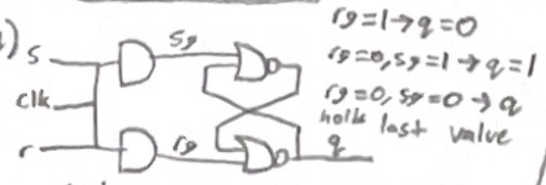


r=1 → q=0
r=0, s=1 → q=1
r=0, s=0 → q holds last value

Gated RS Latch



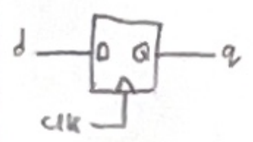
$q_{new} = \bar{r} \wedge (s \vee q_{old})$ ↓equivalent



sg=1 → q=0
sg=0, ss=1 → q=1
sg=0, ss=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



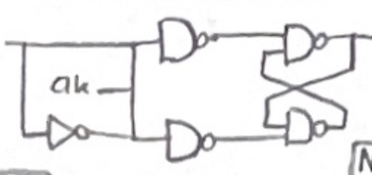
- q = d on clk rising edge

n-bit register?

- Flip flop for each bit all connected to same clk

```
module vDFF(clk, in, out);
  parameter n = 1;
  input clk;
  input [n-1:0] in;
  output [n-1:0] out;
  reg [n-1:0] out;
  always @(posedge clk)
    out = in;
endmodule
```

D Latch



clk=high → q = D
clk=low → q, maintain

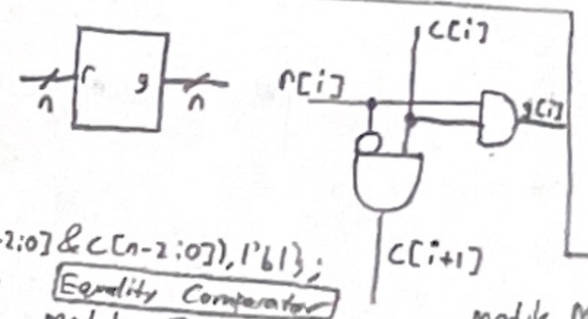
Non Blocking Assignment

- 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
```



Decoder

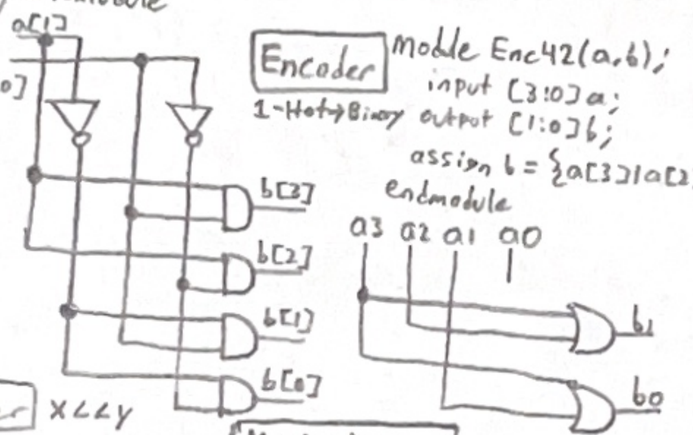
```
module Dec(a, b);
  parameter n = 2;
  parameter m = 4;
  input [n-1:0] a;
  output [m-1:0] b;
  wire [m-1:0] b = 1 << a;
endmodule
```

Binary → 1-Hot

Encoder

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

1-Hot → Binary



Shifter

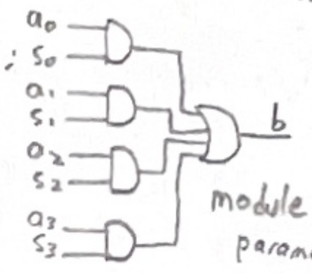
x << y

- Shift x to the left by Y bit positions

Multiplexer

4-input k-wide mux

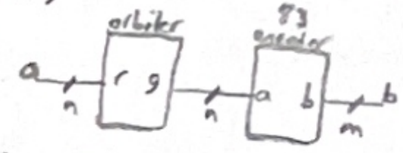
```
module Mux4(a3, a2, a1, a0, s, b);
  parameter k = 1;
  input [k-1:0] a0, a1, a2, a3;
  input [3:0] s;
  output [k-1:0] b;
  reg [k-1:0] b;
  always @(*) begin
    case(s)
      4'b0001 : b = a0
      ⋮
      4'b1000 : b = a3
      default : b = {k{1'bx}};
    endcase
  end
endmodule
```



Priority Encoder

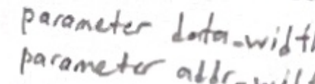
```
module PE(r, b);
  input [7:0] r;
  output [2:0] b;
  wire [7:0] g;
  Arb #(8) a(r, g);
  Enc83 e(g, b);
endmodule
```

Finds position of first 1 in a and outputs binary of that position



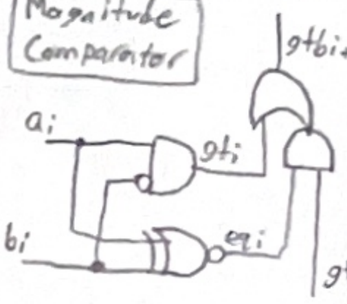
Equality Comparator

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



Magnitude Comparator

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

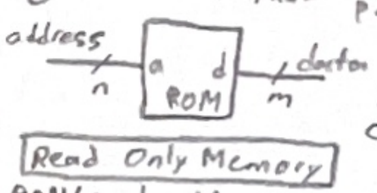



```
module VDFFE(clk, en, in, out);
  parameter n = 1;
  input clk, en;
  input [n-1:0] in;
  output [n-1:0] out;
  reg [n-1:0] out;
  wire [n-1:0] next_out;
  assign next_out = en ? in : out;
  always @(posedge clk)
    out = next_out;
endmodule
```

Register with load enable

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
```



2's Complement

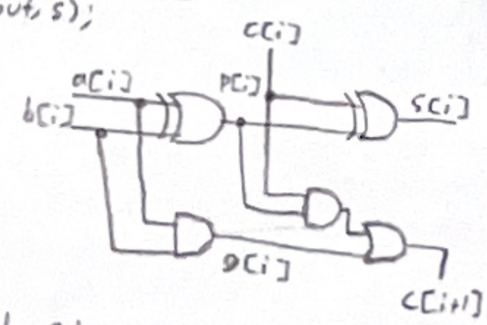
Steps:

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	0	1	1
0	0	1	0	2
0	0	1	1	3

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
```



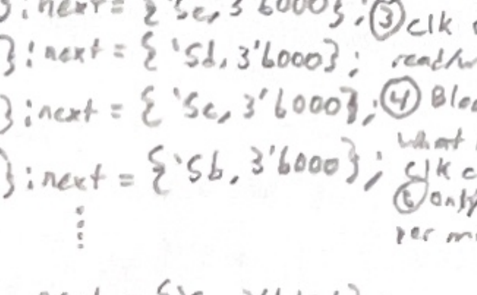
Pipeline Cycles

LDR r1, [r2, #4]
STR r1, [r1, #8]
ADD r3, r3, r1

F	D	X	M	W			
	F	D	S	x	M	W	
		F	D	S	x	M	W

Shifter

```
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 = {`Sc, 3'b000};
      {2'b01, `Sb} : next = {`Sd, 3'b000};
      {2'b11, `Sb} : next = {`Sc, 3'b000};
      {2'bxx, `Sb} : next = {`Sb, 3'b000};
      ⋮
      default : next = {`Sa, 3'b101};
    endcase
  end
  assign {state_next, out} = next;
endmodule
```



Purely Synchronous

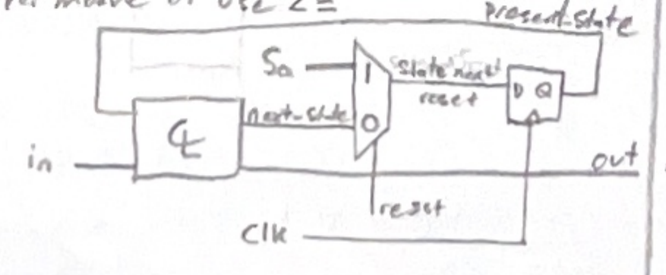
1. only clk in sensitivity list
2. only signals that change on the same edge in the same always block
3. clk should not be read/written inside
4. Block can only describe what happens in 1 clk cycle
5. Only one purely sync per module or use <=

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)



```
module RAM(clk, read_address, write_address, write, din, dout);
  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
```

```
module Addsub(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
```

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

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:
```

Slide Set 8

N-Type: drain, gate, source — ①Gate=0V→open ②Gate=2.5V→closed

P-Type: source, gate, drain — ①Gate=0V→closed ②Gate=2.5V→open

PFET→Pull-up
NFET→Pull-down

NOT Gate: 2.5V, in, out

NAND Gate: a, b, f

NOR Gate: a, b, f

2.6V, 2.5V: a, b, f

Tri-State Inverter: e, a, f

- used if want to drive single wire from two or more places

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

```
assign f = e ? ~a : 1'bz;
```

Slide Set 9

Instruction Encoding

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] → E5905000
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 → E0826CA5
ADD r7, r0, r6, LSL #2 → E0807106
RSB r9, r8, #0 → E2689000

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 r5,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
↓equivalent
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 (store 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
↓equivalent
STR r3, [r2]
ADD r2, r2, #4
```

Spill/Fill
```
SUB sp, sp, #16
STR r6, [sp, #12]
STR r5, [sp, #8]
STR r4, [sp, #4]
STR r1, [sp, #0]
PUSH {r4-r6, LR}
```
```
LDR r1, [sp, #0]
LDR r4, [sp, #4]
LDR r5, [sp, #8]
LDR r6, [sp, #12]
ADD sp, sp, #16
POP {r4-r6, LR}
```

Branch Target Encoding
$PC_{Target} = PC_{Branch} + 8 + 4 * \text{"Target address"}$

Arrays
```
ARRAY:
  .Word 0
  .Word 1
  i
  .Word 9
LDR r1, =ARRAY
```

Slide Set 10

How? → Problems → Solutions

Interrupt Service Routine

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

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

Register Banks
- Different modes
- Blue is the same exact physical register
- orange is a different physical register

CPSR → Current Program Status register

Ex Vs Inter
- interrupt is one form of exception but there are other types ie: memory violation, overflow

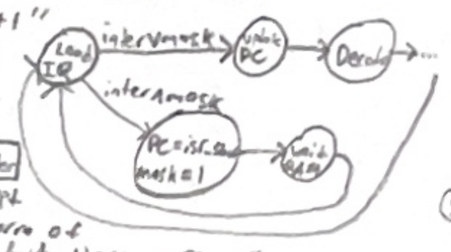
Exception Handling
- Exception occurs, Copies CPSR into SPSR

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

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

Solutions
- Add a Mask register we know there 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

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



Slide Set 11

Fixed Point

0	1	1	0	0	1	1	1
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

= 6.4375

Error
Absolute error = |V(R(x)) - x|
Relative error = |V(R(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
last S or D means double or single precision
FADDS → 32 bit Addition
FSUBS → 32 bit Subtract
FMULS → 32 bit multiply
FDIVS → 32 bit Division

Slide Set 12

x is n times faster than y (Speedup)

$n = \frac{\text{performance}(x)}{\text{performance}(y)}$

$n = \frac{\text{execution}(y)}{\text{execution}(x)}$

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

Cache on Write
① Write through, Store both cache and memory on every byte. Conservative and slow
② Write 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)
$CPI = \frac{\text{Cycle Count}}{\text{Instruction Count}}$

Memory
SRAM: Uses 6 MOS transistors to store a bit, Slow with large systems
DRAM: Uses 1 transistor and a Capacitor to store a bit. Cheaper than SRAM, longer than SRAM (need to refresh due to capacitor leaking, and keep power on)
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
Set Associative:
- Index selects set of blocks as opposed to 1. combo of direct and fully Associative

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

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

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

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

Pipeline Hazards
① Structural, two or more instructions want the same hardware
Sol'n → second instruction has to wait
Sol'n → replicate the hardware block
② Data, can't send info back in time
Sol'n → forwarding, add wires that get info from further back in the pipeline
③ Control, how do you know the next instruction is right
Sol'n → option 2 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 | 8 | 23 |
| X | xxxxxxxx | xxxx....xxxx |
| sign | 127+e | mantissa |

263.3 in 32-Bit FP

2 | 263	
2 | 131	1
2 | 65	1
2 | 32	1
2 | 16	0
2 | 8	0
2 | 4	0
2 | 2	0
2 | 1	0
2 | 0	1

263: 100000111
0.3: 0100110011.....
SN: 1.00000111.2^8
∴ 263.3

+/-	exp	mantissa
= 0	10000111	00000111010011...

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

Amdahl's Law

$\text{Speedup} = \frac{\text{Old Execution time}}{\text{new Execution time}}$

$\text{ExTime}_{new} = \text{ExTime}_{old}[(1 - \text{fractionEnhanced}) + \frac{\text{FractionEnhanced}}{\text{Speedup Enhanced}}]$

Can't be enhanced | Could be enhanced

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

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

GPU = Graphics Processing Unit

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

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 → 000100001100
tag = 0x10
Block offset = 0x4
Block offset = $\log_2$(block size)

State Machine Controller

Instructions

Case 2: opcode = 110
op = 10

How to Execute each instruction

① Put Sximm8 into Rn

How:

- Set vsel to 0100
- Set write to 1
- set nsel to 100

} S1a

② Shift Rm based on the [2:0] Shift and put into RD

How:

- Set nsel to 001
- Set load a to 0
- set load b to 1

} S2a

- set bsel to 0
- Set asel to 1
- Set load c to 1
- set load s to 1

} S2b

- Set write to 1
- Set vsel to 0001

} S2c

③ Add shifted Rm with Rn and put in Rd

How:

- Set nsel to 001
- set load a to 0
- set load b to 1

} S3a

- Set nsel to 100
- Set load a to 1
- set load b to 0

} S3b

- Set asel to 0
- set bsel to 0
- Set load c to 1
- set load s to 1

} S3c

- Set write to 1
- Set vsel to 0001
- Set nsel to 010

} S3d

Top level:

```
`define S1 110
```

① `define S1a 10
// Execute R[Rn] = sx(im8)

② `define S1b 00
// Execute R[RD] = sh_Rm

```
`define S2 101
```

③ `define S2a 00
// Execute R[Rd] = R[Rn] + sh_Rm

④ `define S2b 01
// Execute status = f(R[Rn] - sh_Rm)

⑤ `define S2c 10
// Execute R[Rd] = R[Rn] & sh_Rm

⑥ `define S2d 11
// Execute R[Rd] = ~sh_Rm

⑤ AND shifted Rm and Rn and put in Rd

How:

- Set nsel to 001
- set load a to 0
- set load b to 1

} S5a

- Set nsel to 100
- set load a to 1
- set load b to 0

} S5b

- Set asel to 0
- set bsel to 0
- set load c to 1
- set load s to 1

} S5c

- set nsel to 010
- set vsel to 0001
- set write to 1

} S5d

⑥ NOT shifted Rm and put in Rd

How:

- set nsel to 001
- set load a to 0
- set load b to 1

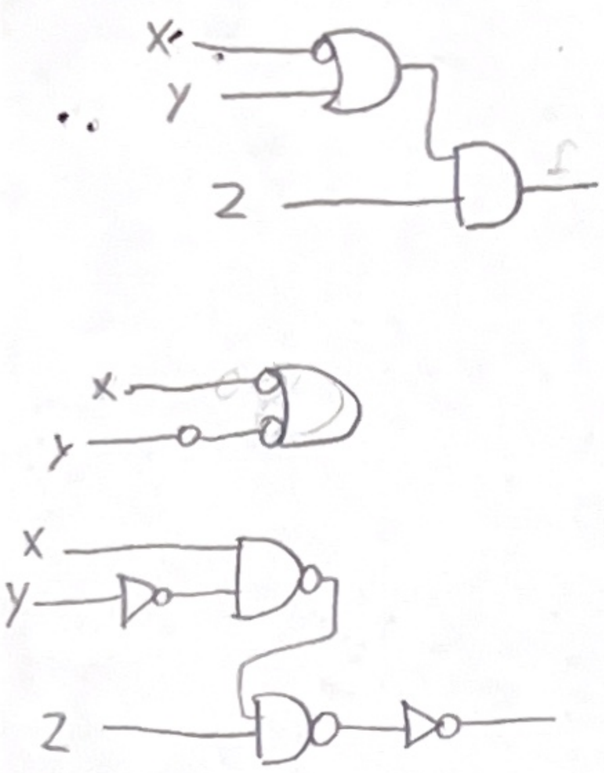
} S6a

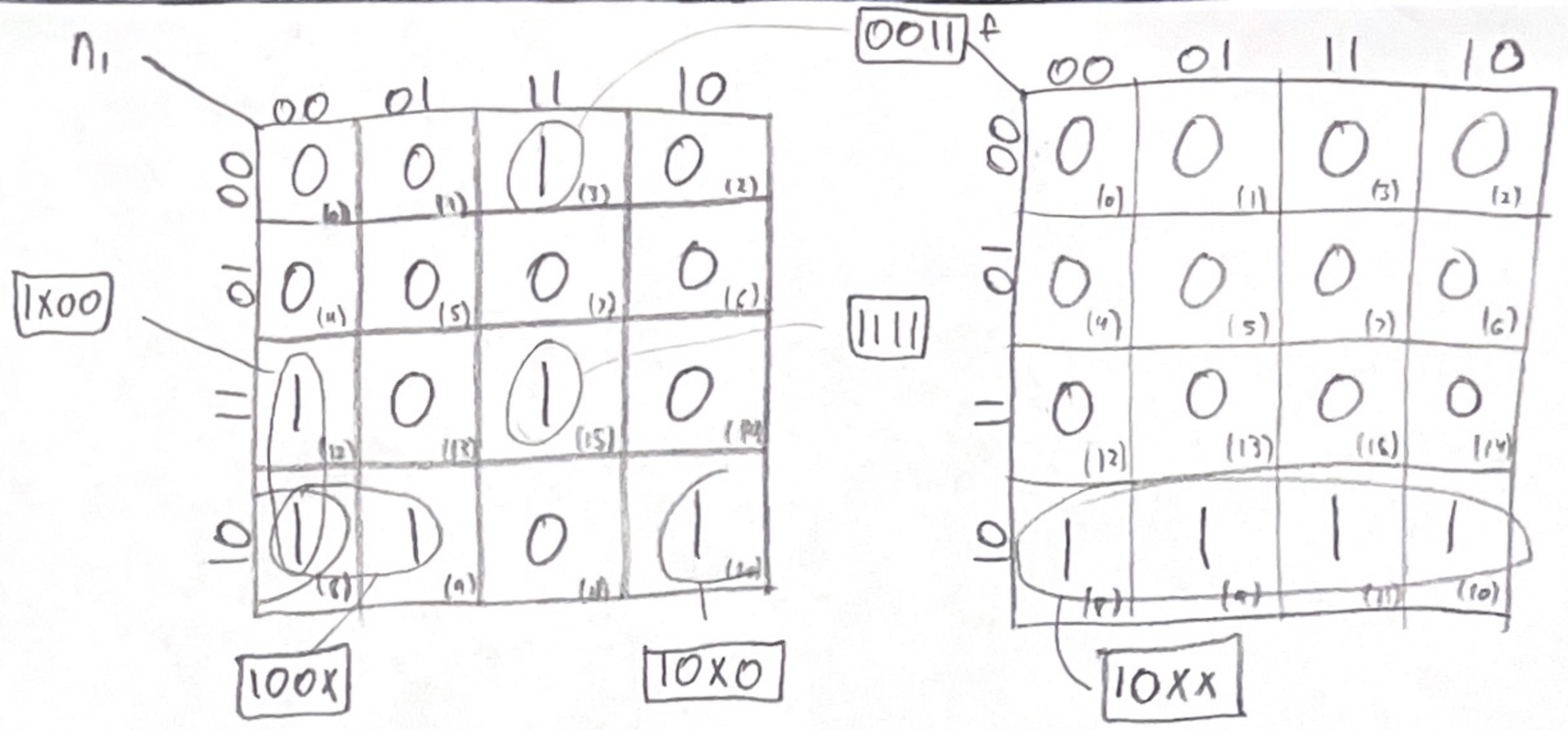
- Set asel to 1
- set bsel to 0
- set load c to 1
- set load s to 1

} S6b

- set nsel to 010
- set vsel to 0001
- set write to 1

} S6c





State encoding

A = 00

B = 01

C = 11

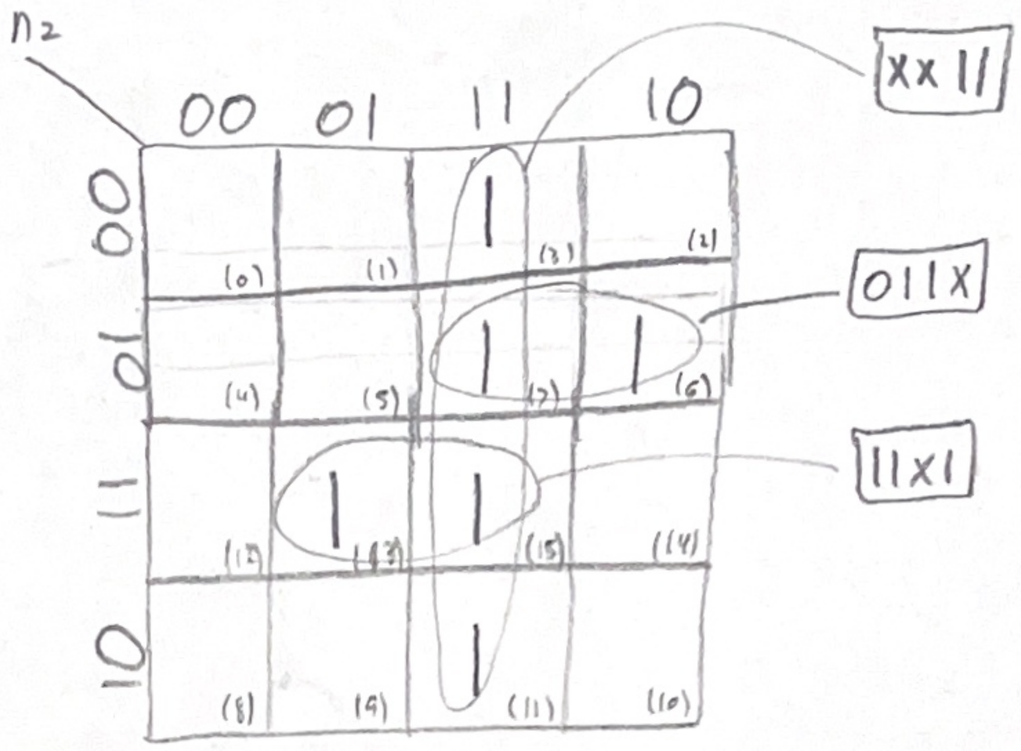
D = 10

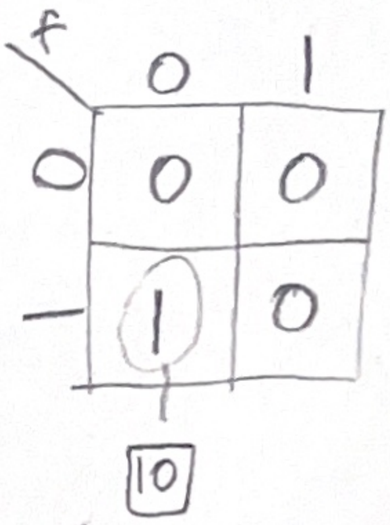
p_2, p_1 = present state

n_2, n_1 = next_state

y, w = in

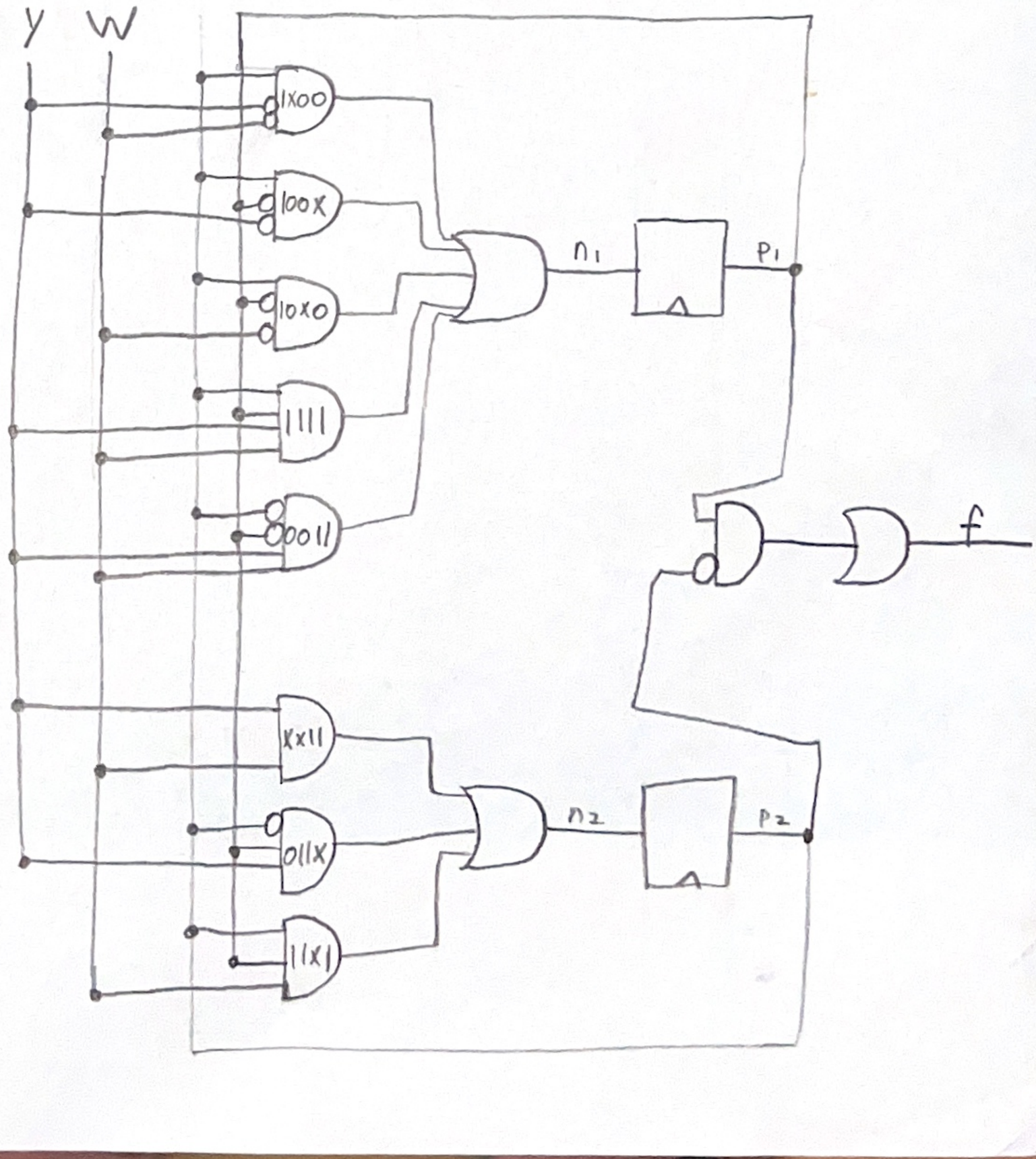
f = out





$f = p_2 \wedge \neg p_1$

	p_2	p_1	y	w	n_2	n_1	f	
A	0	0	0	0	0	0	0	(0)
	0	0	0	1	0	0	0	(1)
	0	0	1	0	0	0	0	(2)
	0	0	1	1	1	1	0	(3)
B	0	1	0	0	0	0	0	(4)
	0	1	0	1	0	0	0	(5)
	0	1	1	0	1	0	0	(6)
	0	1	1	1	1	0	0	(7)
D	1	0	0	0	0	1	1	(8)
	1	0	0	1	0	1	1	(9)
	1	0	1	0	0	1	1	(10)
	1	0	1	1	1	0	1	(11)
C	1	1	0	0	0	1	0	(12)
	1	1	0	1	1	0	0	(13)
	1	1	1	0	0	0	0	(14)
	1	1	1	1	1	1	0	(15)



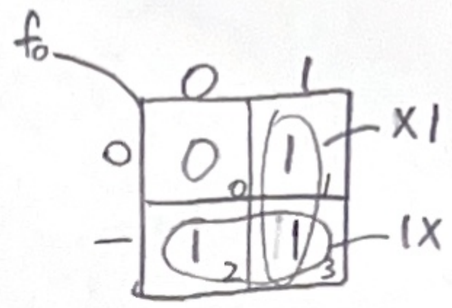
State encoding

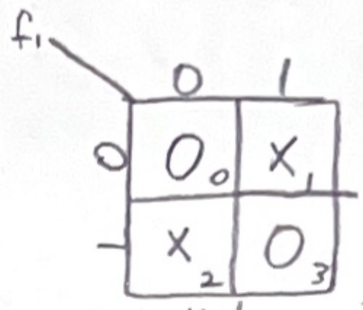
A = 00
B = 01
C = 11
D = 10

let:

p_1, p_0 = present state
n_1, n_0 = next state
X_2, X_1 = inputs
f_1, f_0 = outputs

p_1	p_0	f_1	f_0	
0	0	0	0	(0)
0	1	1	X	(1)
1	0	1	X	(2)
1	1	1	0	(3)



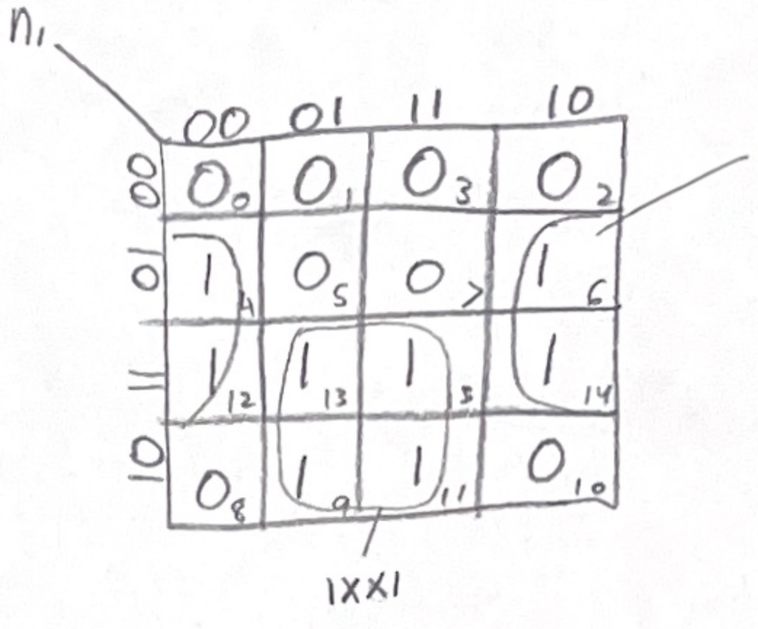


"doesn't depend on present

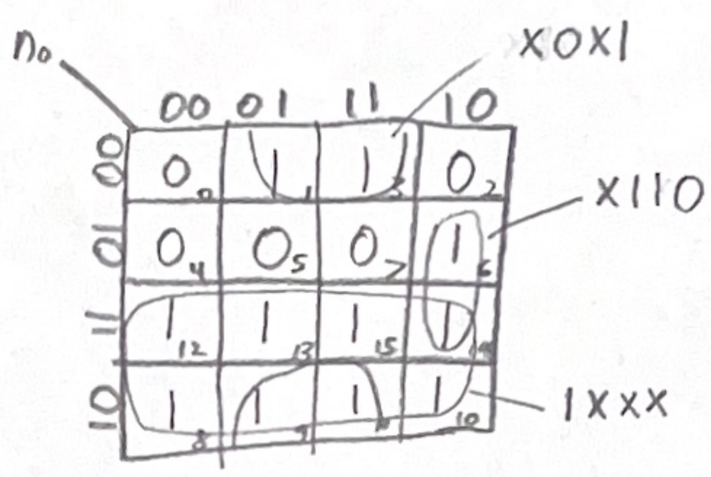
p_1	p_0	f_0
0	0	0
0	1	1
1	0	1
1	1	1

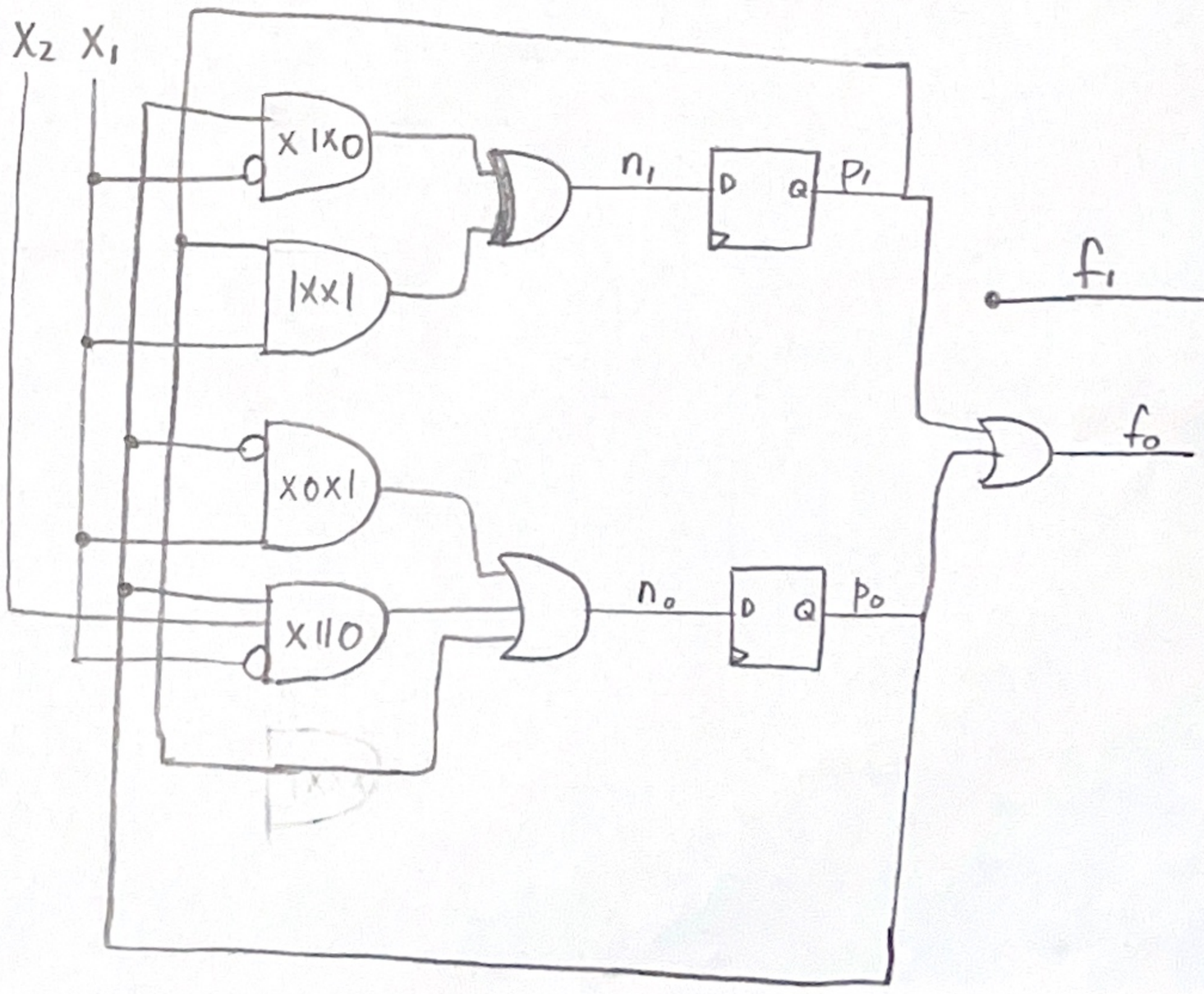
X1X0

this is an OR Gate



	p_1	p_0	X_2	X_1	n_1	n_0	f	f_0	
A	0	0	0	0	0	0			(0)
	0	0	0	1	0	1			(1)
	0	0	1	0	0	0			(2)
	0	0	1	1	0	1			(3)
B	0	1	0	0	1	0			(4)
	0	1	0	1	0	0			(5)
	0	1	1	0	1	1			(6)
	0	1	1	1	0	0			(7)
D	1	0	0	0	0	1			(8)
	1	0	0	1	1	1			(9)
	1	0	1	0	0	1			(10)
	1	0	1	1	1	1			(11)
C	1	1	0	0	1	1			(12)
	1	1	0	1	1	1			(13)
	1	1	1	0	1	1			(14)
	1	1	1	1	1	1			(15)



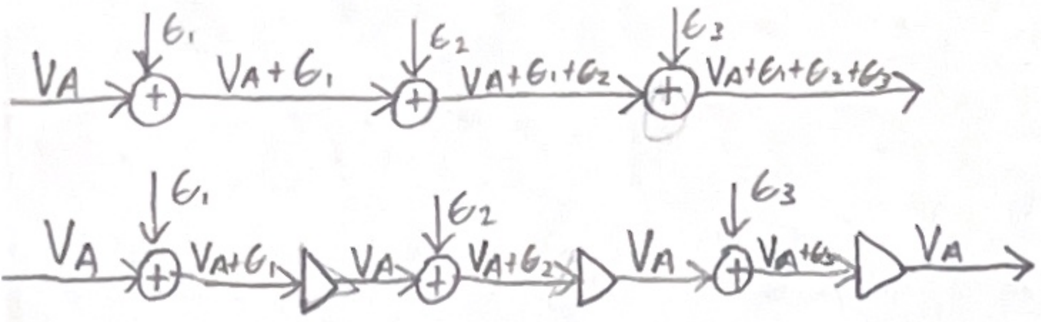


CPEN 211

Chapter 1: Digital Abstraction

Noise

ANALOG (Noise accumulates)



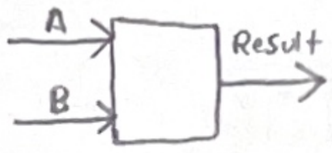
DIGITAL

▷ Buffer: determines if its in a 0 or 1 range and makes it a pristine 0 or 1. Effectively scrubbing noise.

Digital Signals

- N elements requires $\log_2 N$ bits. ie: 8 colours needs $\log_2 8 = 3$ bits

Combinational Logic Circuit



output = f(input)

- no memory
- result determined by present inputs

Sequential Logic Circuit

- Has memory

Verilog

- Verilog is a hardware description language HDL NOT code/software
- Quartus does not run verilog it runs the logic gates made from the verilog

Chapter 3: Boolean Algebra

Axioms

$1 \wedge X = X$ $\quad 0 \wedge X = 0$

$1 \vee X = 1$ $\quad 0 \vee X = X$

$\bar{0} = 1$ $\quad \bar{1} = 0$

Equivilantly

$\neg 0 = 1$ $\quad \neg 1 = 0$

Properties

Commutative: $X \wedge Y = Y \wedge X$

Associative: $X \wedge (Y \wedge Z) = (X \wedge Y) \wedge Z$

Distributive: $X \wedge (Y \vee Z) = (X \wedge Y) \vee (Y \wedge Z)$

Idempotence: $X \wedge X = X$; $X \vee X = X$

Complementation: $X \wedge \bar{X} = 0$; $X \vee \bar{X} = 1$

Absorbtion: $X \wedge (X \vee Y) = X$

Combining: $(X \wedge Y) \vee (X \wedge \bar{Y}) = X$

De Morgans: $\overline{(X \wedge Y)} = \bar{X} \vee \bar{Y}$

Truth Tables

a	b	a∧b	a∨b
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	1

a	ā
0	1
1	0

Dual Functions

- Swith everything

$f(x,y) = (x \wedge 1) \vee (0 \vee \bar{y})$

$f^0(x,y) = (x \vee 0) \wedge (1 \wedge \bar{y})$

Duality

- You can change every $\wedge \to \vee$ and $\vee \to \wedge$ and $0 \to 1$ and $1 \to 0$ and everything holds

Normal Form

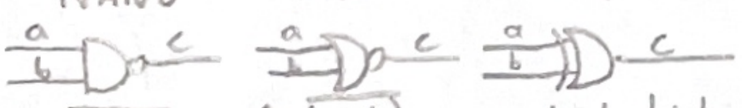
a	b	c	q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

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

Logic Gates

AND OR NOT

NAND NOR XOR

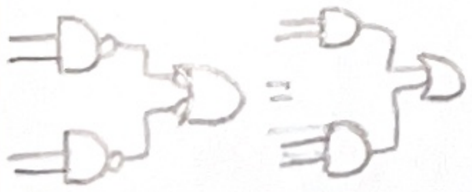


$f = \overline{(a \wedge b)}$ $\quad f = \overline{(a \vee b)}$

- output high if one input is high
- output true if odd number of inputs is high

De Morgan

Bubble Rule



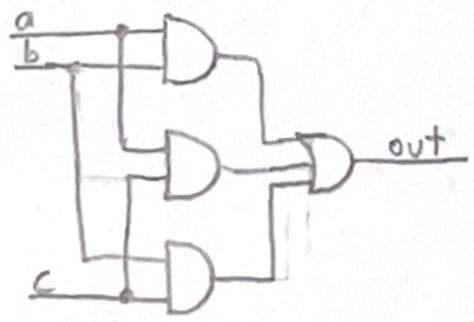
a	b	XOR
0	0	0
1	0	1
0	1	1
1	1	0

Verilog

AND OR NOT XOR

& | ~ ^

```
module Majority(a, b, c, out);
  input a,b,c;
  output out;
  assign out = (a&b)|(a&c)|(b&c);
endmodule
```



Chapter 6: Combinational Logic

FF
- no memory

Binary

2^n = combos
where n is bits

$2^3 = 8$

$2^4 = 16$

closed
- closed under acyclic composition
- you can add them up and its still combinational as long as you don't make loops

minterm
- Product term

$f = \sum_{dcba} m(1,2,3,5,7,11,13)$

- must have min 4 if dcba

Normal Form
- Sum of the product terms

Implicant
- a minterm could have value 0
- an implicant could have less terms by combining

Karnaugh Map

	00	01	11	10
00	0	1	1	1
01	0	1	1	0
11	0	1	0	0
10	0	0	1	0

Grouping
- Can group in
  - 8
  - 4
  - 2
- must choose biggest and work down

Don't Cares
- allows you to cover a larger area with X's and thus make less implicants

Prime implicant
- an implicant that cannot be made larger
- unique
- shares no bits

Essential Prime implicant
- an implicant with a 1 only covered once

Chapter 7 Verilog

"To describe combinational logic to be synthesized we only use assign and case statements"

Structure

```
module Count(in, out);
  input [3:0] in;
  output [out;
  <Module Body (logic)>
endmodule
```

Wire
- outputs are a wire by default
- Assigned by an ASSIGN statement
- Used to connect modules

Reg
- If assigned in a case statement or an always block
- Not a register

Case Statement
- specifys a truth table
- must have all possibilities covered or include a default

Always Block

```
always@(in) begin
end
```

```
always@(*) begin
end
```

Begin/End
- like { } in C (Brackets)

Assign

Computed whenever RHS changes.
Describes hardware

Testbench

Initial → like an always block
#100 → wait 100ps
$display("like printf");
repeat → loop generator

===
!==

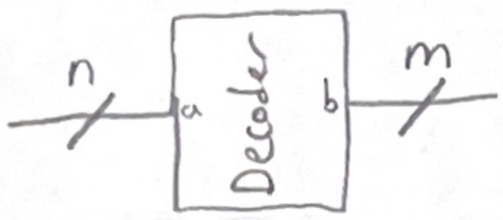
Chapter 8 Combinational Building Blocks

Decoder

- Converts one type of symbol to a different type

In general:

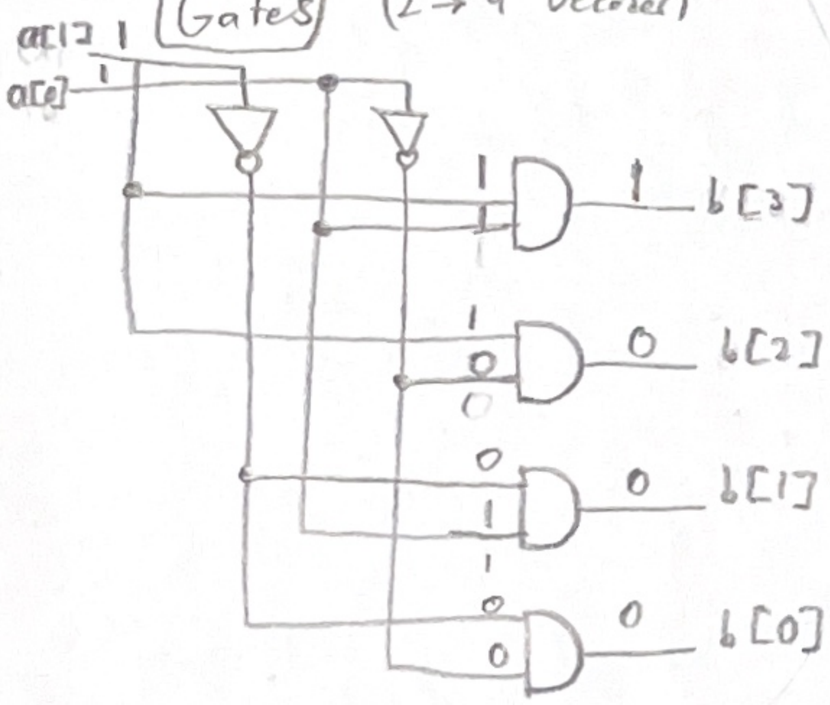
Decoder: Binary → One-hot
Encoder: one-hot → binary



Verilog (Decoder)

```
// n -> m decoder
// a - binary input (n bits wide)
// b - one-hot output (m bits wide)
module Dec(a,b);
    parameter n=2;
    parameter m=4;

    input [n-1:0] a;
    output [m-1:0] b;

    assign b = 1<<a;
endmodule
```

Gates (2 → 4 Decoder)



Key Building Blocks

1. Decoder
2. Encoder
3. Multiplexer
4. Arbiter
5. Comparator
6. Read only memories

Paramater

- Can override default values

Dec #(3,8) dec38(a,b);

Encoder

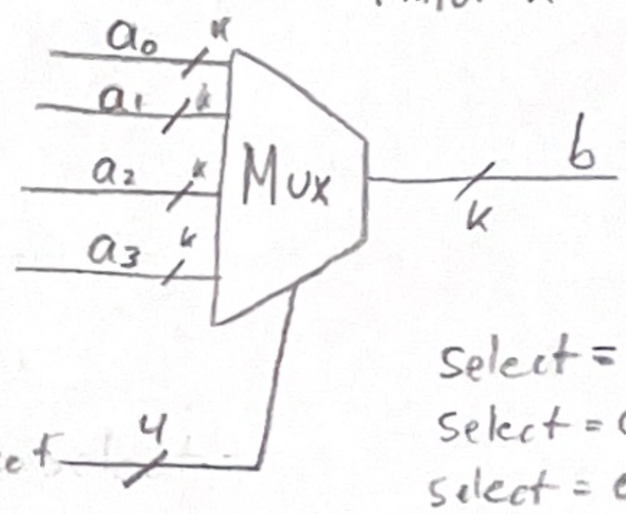
- Converts one-hot to binary


```
module Enc42(a,b);
    input [3:0] a;
    output [1:0] b;

    assign b= {a[3]|a[2], a[3]|a[1]};
endmodule
```

Multiplexer

- Chooses what input becomes the output

4 input k bit mux, one-hot select

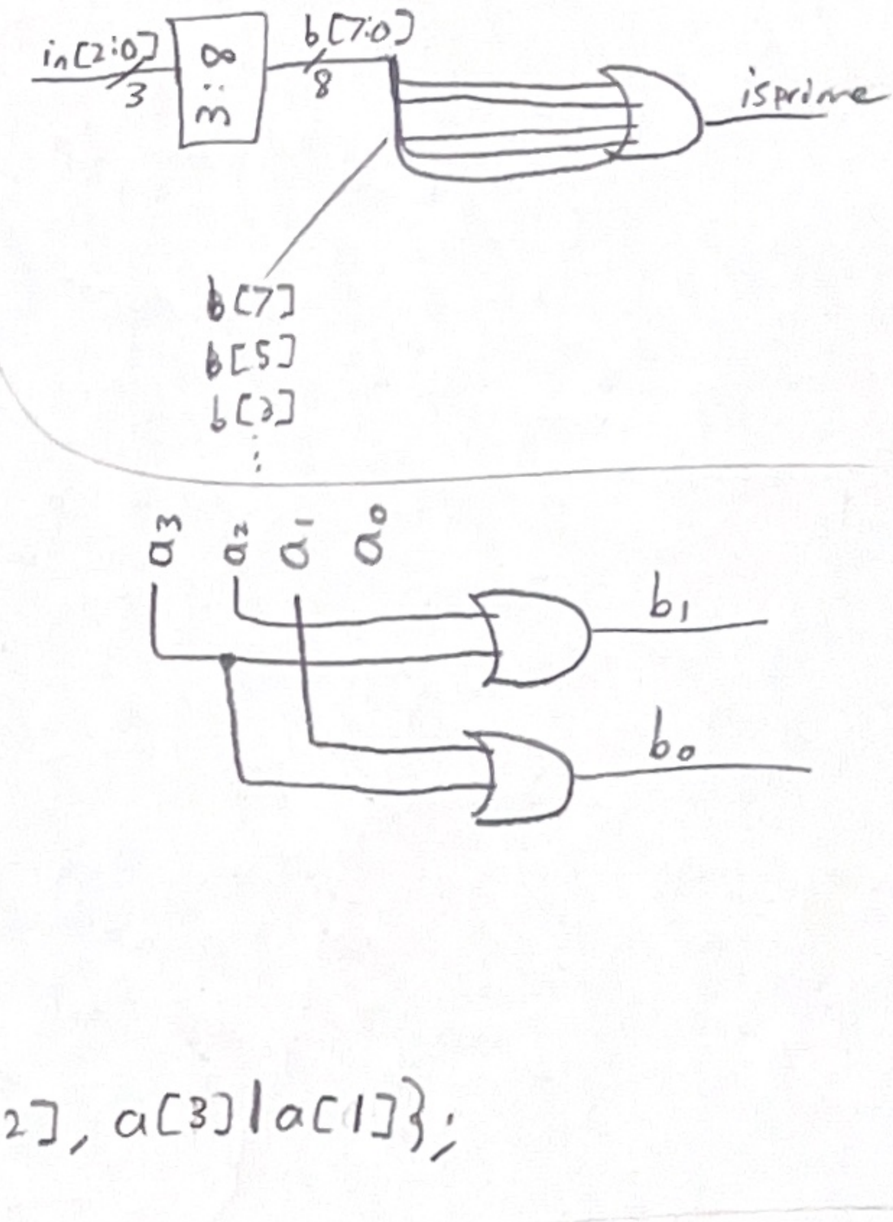


Select = 0001	b = a0
Select = 0010	b = a1
Select = 0100	b = a2
Select = 1000	b = a3

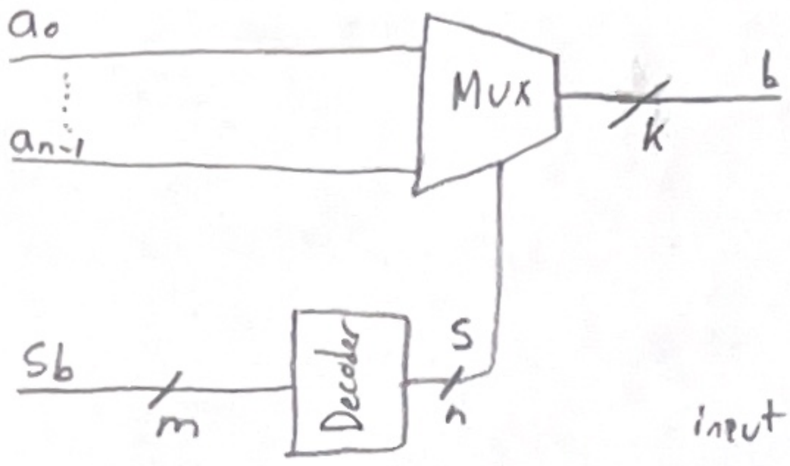
```
module Mux4(a3,a2,a1,a0,s,b);
    parameter k=1;
    input [k-1:0] a0,a1,a2,a3;
    input [3:0] s;
    output [k-1:0] b;
    wire [k-1:0] b;

    assign b = ({k{s[0]}} & a0 |
                ({k{s[1]}} & a1 |
                ({k{s[2]}} & a2 |
                ({k{s[3]}} & a3);
endmodule
```

Example (Prime number function)

```
module Primed(in, isprime);
    input [2:0] in;
    output      isprime;
    wire [7:0] b;

    wire isprime = b[1] | b[2] | b[3] | b[4] | b[5] | b[6] | b[7];

    Dec #(3,8) d(in,b);
endmodule
```



Binary-Select Mux



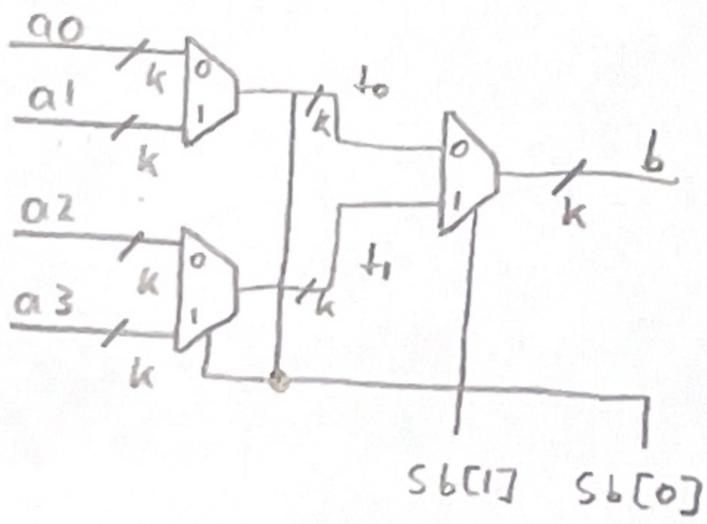
-Less efficient

```
module Muxb3(a2,a1,a0,Sb,b);
    parameter k=1;
    input [k-1:0] a2,a1,a0
    input [1:0] Sb;
    output[k-1:0] b;
    wire [2:0] s;

    Dec #(2,3) d(Sb,S);
    Mux3 #(k) m(a2,a1,a0,s,b);
endmodule
```

Factoring

4:1 Mux Binary Select
Built out of 2 smaller Muxes

```
module Mux4b(a3,a2,a1,a0,sb,b);
    parameter k=1;
    input [k-1:0]a3,a2,a1,a0;
    input [1:0]sb;
    output [k-1:0]b;
    wire [k-1:0] t0,t1;

    assign t0 = sb[0] ? a1 : a0;
    assign t1 = sb[0] ? a3 : a2;
    assign b = sb[1] ? t0 : t1;
```



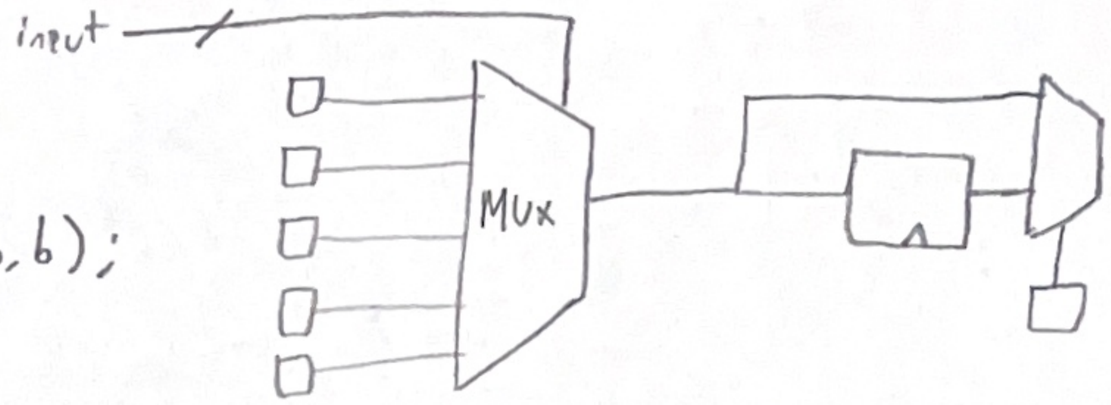
Shannon Expansion

$F(X_0, X_1, X_2) = (\overline{X_0} \wedge F(0, X_1, X_2)) \vee (X_0 \wedge F(1, X_1, X_2))$

-used to simplify muxes to smaller muxes

-Can be done with any X input

FPGA Logic Block



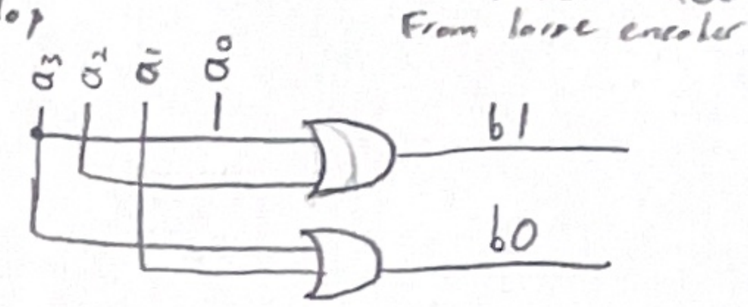
lookup table

Flip Flop

4:2 Encoder

```
module Enc42(a,b,c);
    input [3:0] a;
    output [1:0] b;
    output c;
    wire [1:0] b;
    wire c;

    assign b[1] = a[3] | a[2];
    assign b[0] = a[3] | a[1];
    assign c = |a;  // c = a[0] | a[1] | a[2] | a[3];
endmodule
```

* c values = Blue
* D values = Red
From loose encoder example



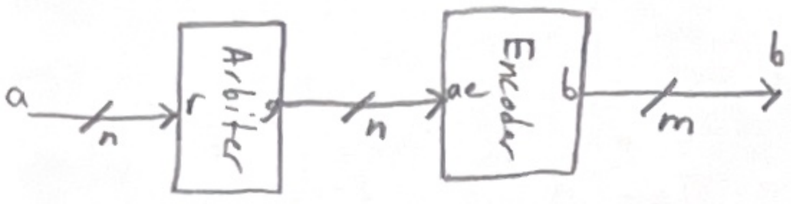
Arbiter

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

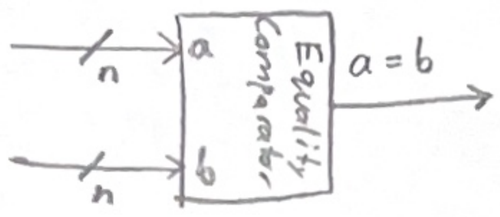
Priority Encoder

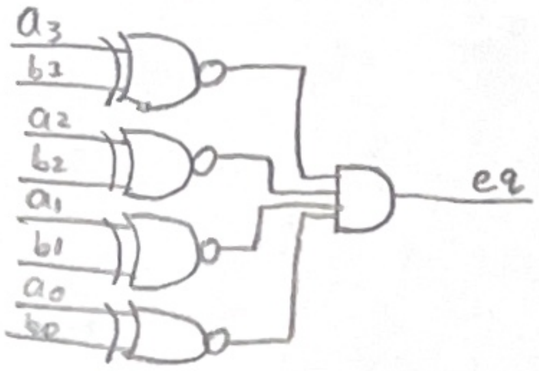
-Finds the first occurrence of a 1 in the input bus starting at the LSB and outputs its position in binary


```
module PriorityEncoder83(r,b);
  input [7:0] r;
  output [2:0] b;
  wire [7:0] g;
  Arb #(8) a(r,g);
  Enc83    e(g,b);
endmodule
```



Equality Comparator



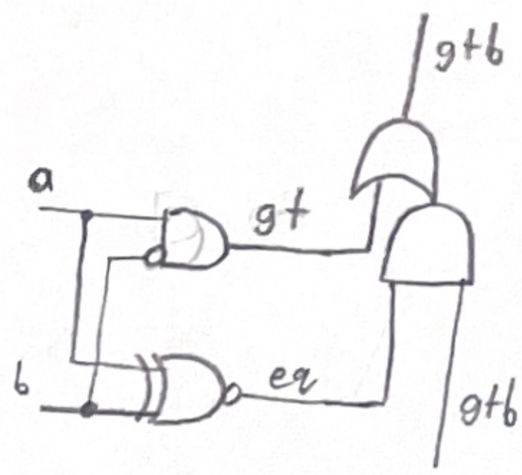



```
module EqComp(a,b,eq);
  parameter k=8;
  input [k-1:0] a,b;
  output eq;
  wire eq;

  assign eq = (a==b);
endmodule
```

Magnitude Comparator


```
module MagComp(a,b,gt);
  parameter k=8;
  input [k-1:0] a,b;
  output gt;
  wire[k-1:0] eqi = a~^b;
  wire [k-1:0] gti = a & ~b;
  wire [k:0]  gtb = {((eqi[k-1:0] & gtb[k-1:0]) | gti[k-1:0]), 1'b0};
  wire  gt = gtb[k];
```

a>b iff a[i]>b[i] AND a[j] = b[j] for j>i

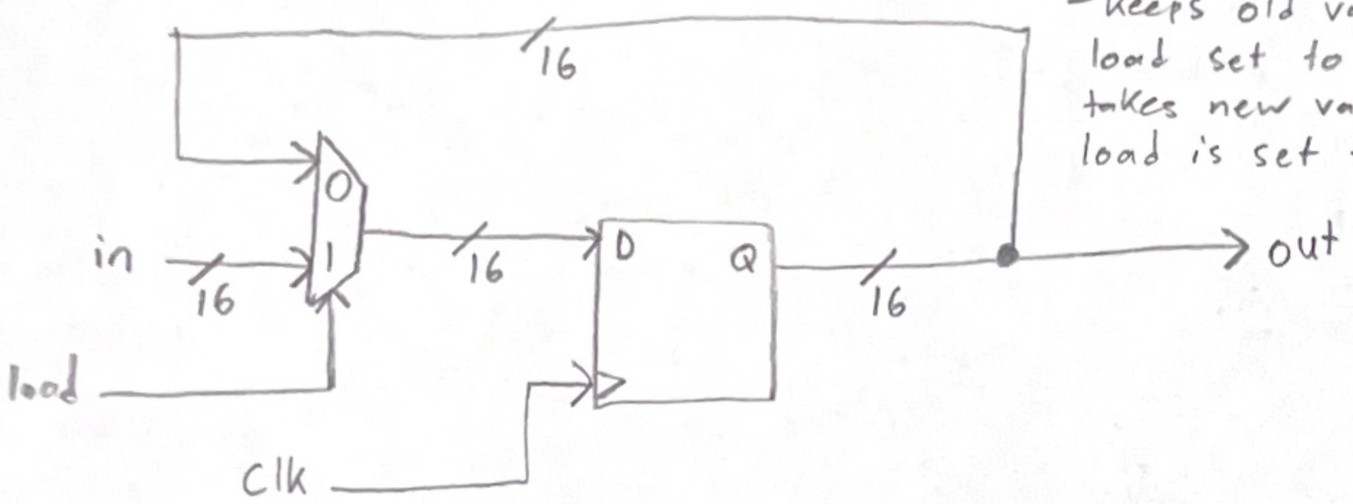



```
module MagComp(a,b,gt);
  parameter k=8;
  input [k-1:0] a,b;
  output gt;
  Wire gt = (a>b);
endmodule
```

Read only Memory (ROM)

```
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**addrwidth-1:0];

  initial $readmemb(filename, mem);
  assign dout = mem(read_address);
endmodule
```

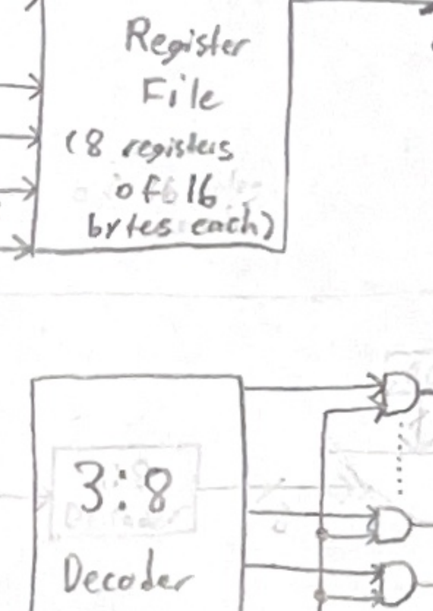
Register with load enable



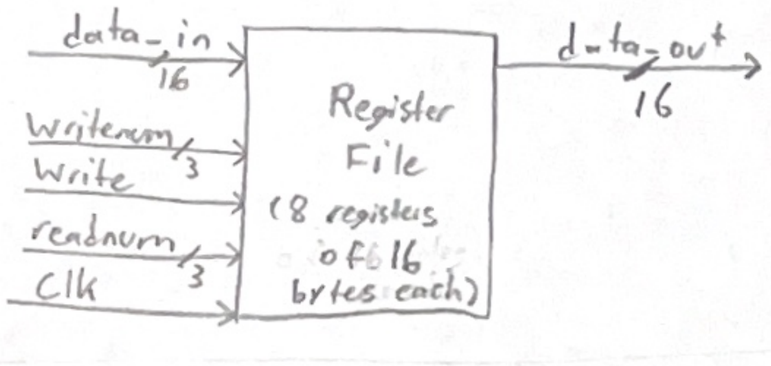
- 16 Flip Flops
- Remembers
- keeps old value if load set to 0 and takes new value if load is set to 1

- Remebers value as long as load set to 0

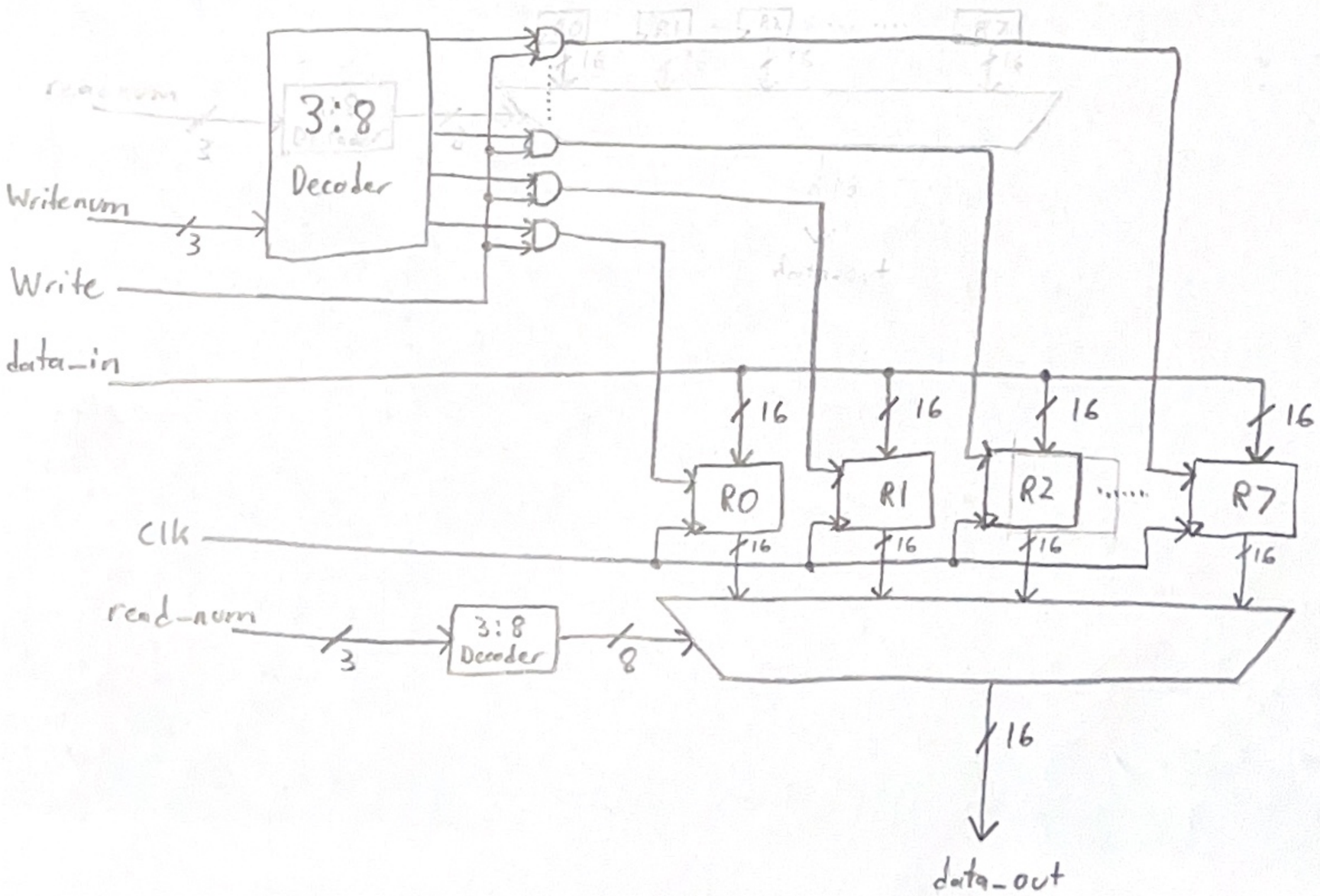
```
module vDFFE(clk, en, in, out);
  parameter n = 1; // width
  input clk, en;
  input [n-1:0] n;
  output [n-1:0] out
  reg   [n-1:0] out
  wire [n-1:0] next_out;

  assign next_out = en? in:out;

  always @(posedge clk)
     out= next_out;
endmodule
```



Register File



- data-in, data_out is values
- Writenum, readnum are addresses



ARM

Levels of Representation

High-Level
```
temp = V[k]
temp++;
```
↓ Compiler

Assembly
```
LDR R5, [R2]
LDR R6, [R2,#4]
```
↓ Assembler

Machine language
```
1100 0101
1111 1000
```
↓

Control signals — ALUop[0:3] ← f(instr Reg[25:20])

Assembly Syntax

Ex C: a = b + c + d + e;

```
ARM: ADD a,b,c //add b and c and place in a
     ADD a,a,d
     ADD a,a,e
```

Ex C: f = (g + h) - (i + j);

```
ARM: ADD t0, g, h
     ADD t1, i, j
     SUB f, t0, t1 // f = t0 - t1
```

Types of Memory

① DRAM (Dynamic)
② SRAM (static)
③ Flash memory (holds value even when power off)

Reading from Memory

```
LDR R5, [R2, #8]   (address)
```
// R2 contains <val> 100, takes that and stores it in R5

#(<val>) is the offset to get to the right address

Bytes

- int takes up 32 bit 'word' broken up into 4-8bit bytes

0000 0000 0000 1010

- ARM puts LSB first in 32 bit 'word' "little endian"
- Could also put MSB first "big endian"

Address	Data
16	
12	
8	
4	10
0	

Writing to Memory

```
STR r6, [r3, #48]
```
// stores the value in r6 at that address

Immediate operands

```
ADD r3, r3, #4
```
// r3 = r3 + 4

Instruction Field

bits

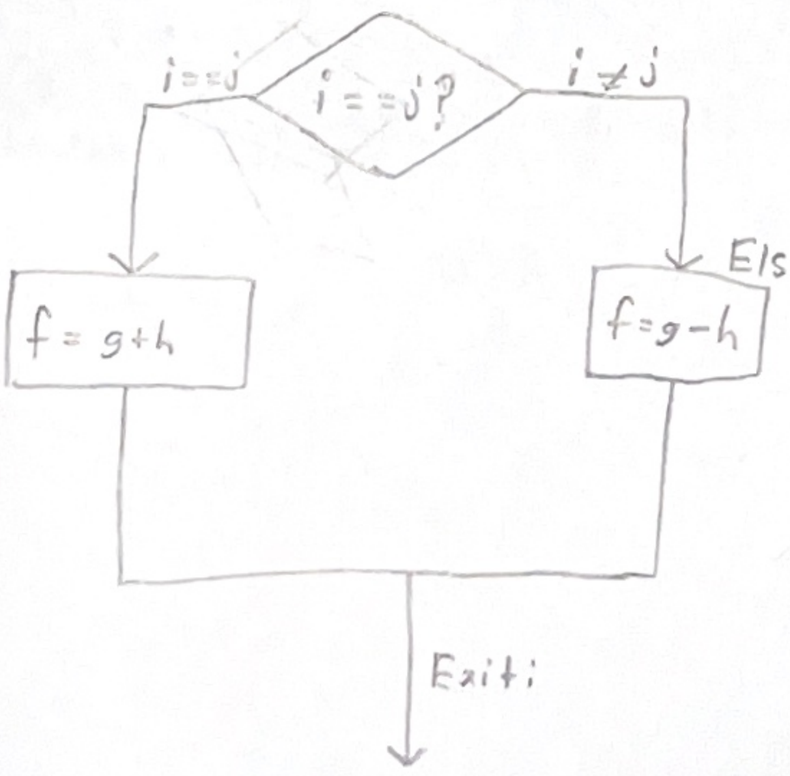
Cond	F	I	Opcode	S	Rn	Rd	Operand 2
4	2	1	4	1	4	4	12

Logical Operations

```
AND r5, r1, r2 // reg r5 = reg r1 & reg r2
ORR r5, r1, r2 // reg r5 = reg r1 | reg r2
MVN r5, r2 // reg r5 = ~reg r2
MOV r6, r5 // reg r6 = reg r5
```

Branches

```
if(i==j)
  f = g+h;
else f = g-h;
```
} C Code




```
CMP R3, R4
BNE Else //Branch not equal, go to Else:
ADD r0, r1, r2 //skipped if i≠j
B Exit
Else: SUB r0, r1, r2
Exit:
```

Loops

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

ARM Assembly
```
Loop: ADD r12, r6, r3, LSL #2
      // r12 = address of save[i]
      LDR r0, [r12, #0]
      //Temp reg r0 = save[i]
      CMP r0, r5
      BNE Exit
      ADD r3, r3, #1
      B Loop
Exit:
```

Conditional Execution

-Can eliminate branches

```
CMP R3,R4                CMP R3,R4
BNE Else                 ADDEQ r0,r1,r2
ADD r0,r1,r2     ⟶      SUBNE r0,r2,r2
  B  Exit
Else: SUB r0,r2,r2
Exit:
```

Hexadecimal

D_{10}	H_{16}
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	a
11	b
12	c
13	d
14	e
15	f

Hexidecimal → Decimal:

$(A\ B\ 0\ 9)_{16} = (43785)_{10}$

$\downarrow\downarrow\downarrow\downarrow$

$16^3\ 16^2\ 16^1\ 16^0$

$10\cdot16^3 + 11\cdot16^2 + 0\cdot16 + 9\cdot1 = 43,785$

Decimal → Hexadecimal:

$(479)_{10} = (1DF)_{16}$

$479 \div 16 = 29$ R15 → F

$29 \div 16 = 1$ R13 → D

$1 \div 16 = 0$ R1 → 1

Step 1

-ARM uses R0-R3 for the first 4 parameters passed

-More than 4 parameters requires using memory

```
main:
    MOV R0, #1
    MOV R1, #5
    MOV R2, #9
    MOV R3, #20
```

Step 2

-ARM uses branch and link BL to transfer control to subroutine

-Places address to return home to in LR (R14)

```
BL leaf_function
```

Step 5

-ARM puts the result to be returned in R0.

*Note this overwrites Parameter 1 placed in R0. This is ok because we will save a copy of it on the stack

```
MOV R0, R4
```

Indirect Branches

```
Bx  Rm

MOV PC, Rm
```

Step 6

```
MOV pc, lr
```

-PC(R15) in ARM

-Value in lr is pc adress at BL+4

Function Call in ARM

Steps:

1. Put parameters where function can find them
2. Transfer control to the function
3. * Acquire Storage for function
4. Perform task
5. Put result where calling function can access it
6. Return control to the point of origin (function might be called from many places)

C Code to Execute

```
/* main.c */
extern int leaf_example(int, int, int, int);
void main(){
    int result = leaf_example(1, 5, 9, 20);
    if (result > 10)
        ⋮
}

/* leaf.c */
int leaf_example(int g, int h, int i, int j){
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

Terminology

Caller: makes the call ie: main

Callee: function being called ie: leaf_example

Convention:

- ARM uses Callee Saving