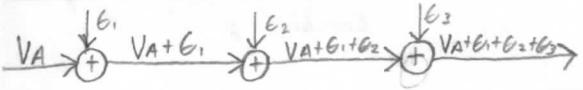
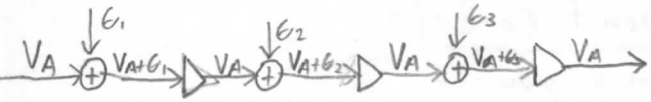
CPEN 211

Chapter 1: Digital Abstraction

Noise

ANALOG (Noise accumulates)





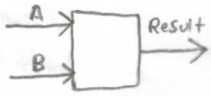
DIGITAL

▷ Buffer: determines if it's 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



- no memory
- result determined by present inputs

output = f(input)

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

$1 \wedge X = X \qquad 0 \wedge X = 0$

$1 \vee X = 1 \qquad 0 \vee X = X$

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

Equivilantly

$\neg 0 = 1 \qquad \neg 1 = 0$

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	$\bar{a}$
0	1
1	0

Duality

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

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: $\overline{(X \wedge y)} = \bar{X} \vee \bar{y}$

Dual Functions

- Swith everything

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

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

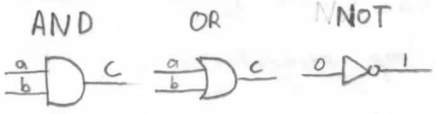
Normal Form

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

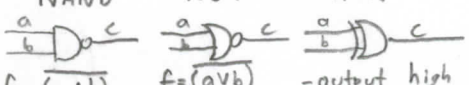
$f(a,b,c) = (\bar{a} \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge \bar{c}) \vee (a \wedge b \wedge c)$

Logic Gates

AND OR NOT



NAND NOR XOR

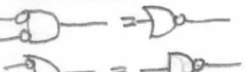


$f = \overline{(a \wedge b)}$ $f = \overline{(a \vee b)}$

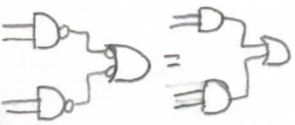
- output high if one input is high
- output true if odd number of inputs is high

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

De Morgan



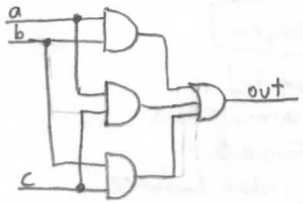
Bubble Rule



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

Q

-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

Don't Cares

-allows you to cover a larger area with X's and thus make less implicants

Grouping

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

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

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

===
!==

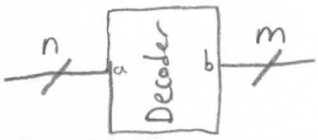
Chapter 8 Combinational Building Blocks

Decoder

- Converts one type of symbol to a different type

In general:

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



Key Building Blocks

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

Paramater

- Can override default values

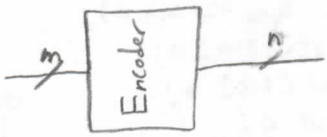
```
Dec #(3,8) dec38 (a,b);
```

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
```

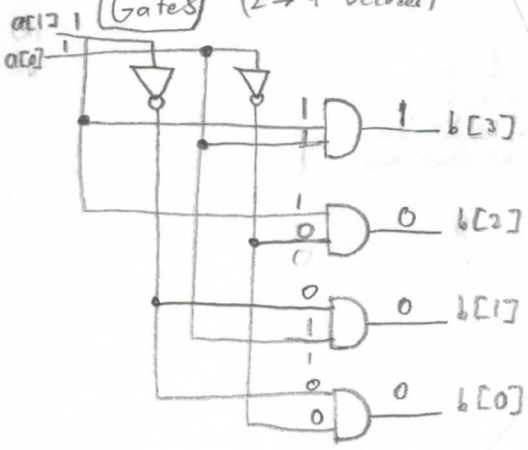
Encoder

- 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
```

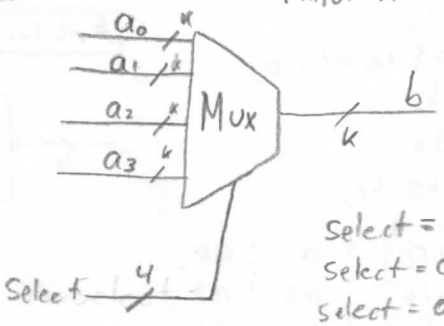
Gates (2 → 4 Decoder)



Multiplexer

- Chooses what input becomes the output

4 input k bit mux, one-hot select

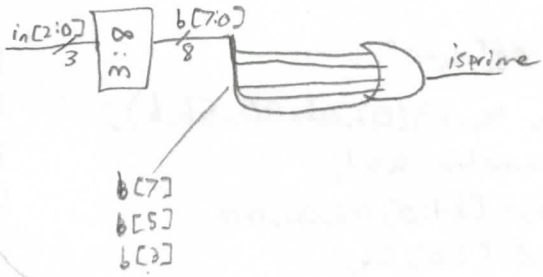


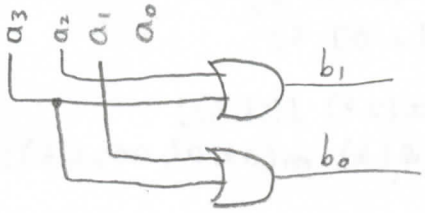
Select = 0001	b = a_0
Select = 0010	b = a_1
Select = 0100	b = a_2
Select = 1000	b = a_3

```
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
```

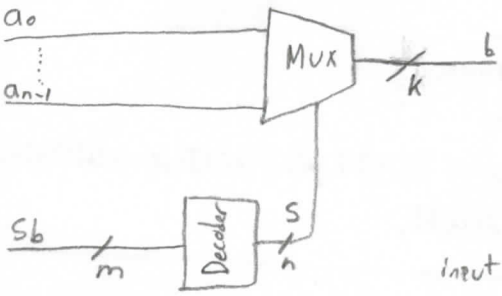
Example (Prime number function)

```
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



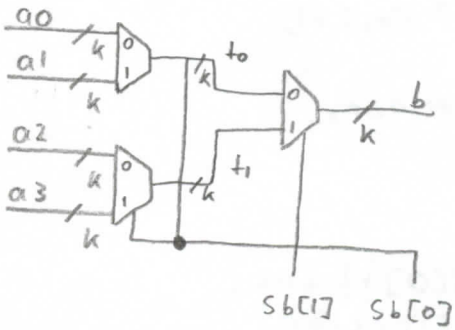
-Less efficient

```
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;
```



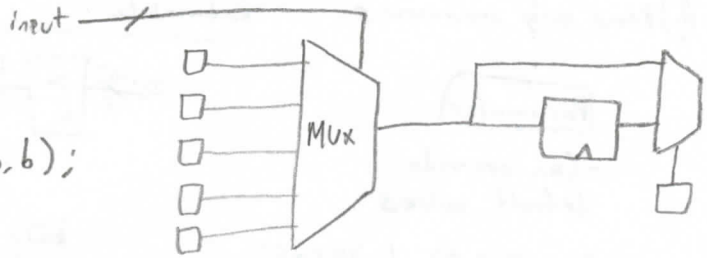
Shannon Expansion

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

-used to simplify muxes to smaller muxes

-Can be done with any X input

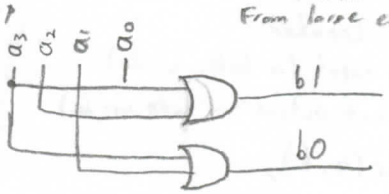
FPGA Logic Block



lookup table

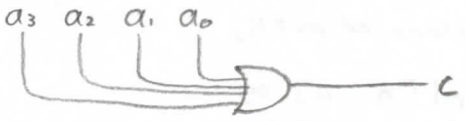
Flip Flop

*C values = Blue
*D values = Red
From large encoder example



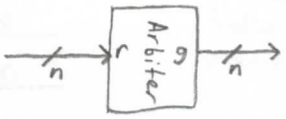
4:2 Encoder

```
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
```



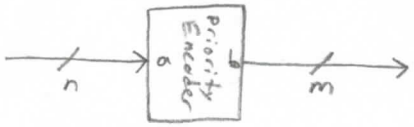
Arbiter

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

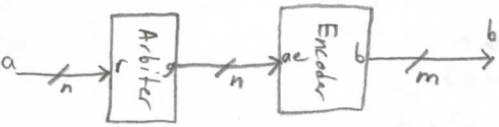


Priority Encoder

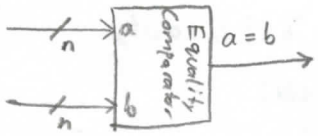
- Finds the first occurrence of a 1 in the input bus starting at the LSB and outputs its position in binary

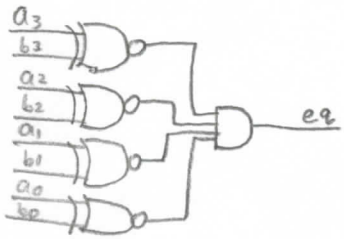



```
module PriorityEncoder83(r,b);
    input [7:0] r;
    output [2:0] b;
    wire [7:0] g;
    Arb #(8) a(r,g);
    Enc83    e(g,b);
endmodule
```



Equality Comparator

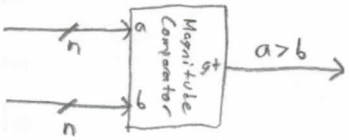


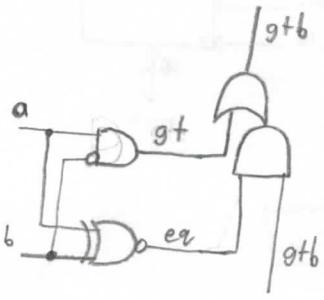



```
module EqComp(a,b,eq);
    parameter k=8;
    input [k-1:0] a,b;
    output eq;
    wire eq;

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

Magnitude Comparator

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 [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];
```

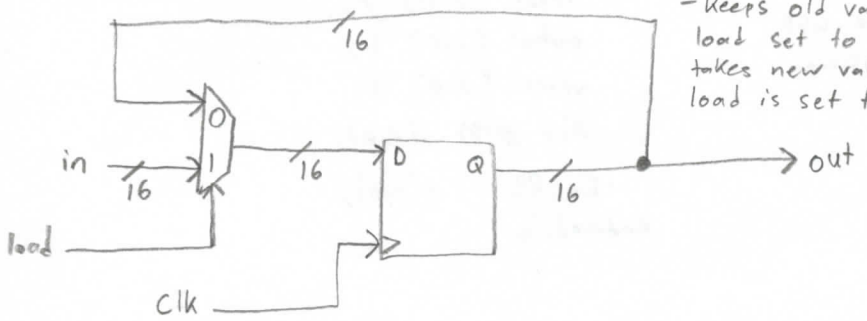
```
module MagComp(a,b,gt);
    parameter k=8;
    input [k-1:0] a,b;
    output gt;
    wire gt = (a>b);
endmodule
```

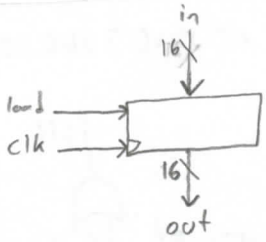
Read only Memory (ROM)

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

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

Register with load enable

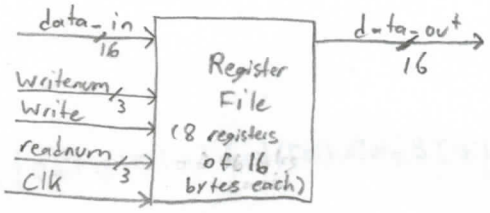
- 16 Flip Flops
- Remembers
- Keeps old value if load set to 0 and takes new value if load is set to 1
- Remembers value as long as load set to 0



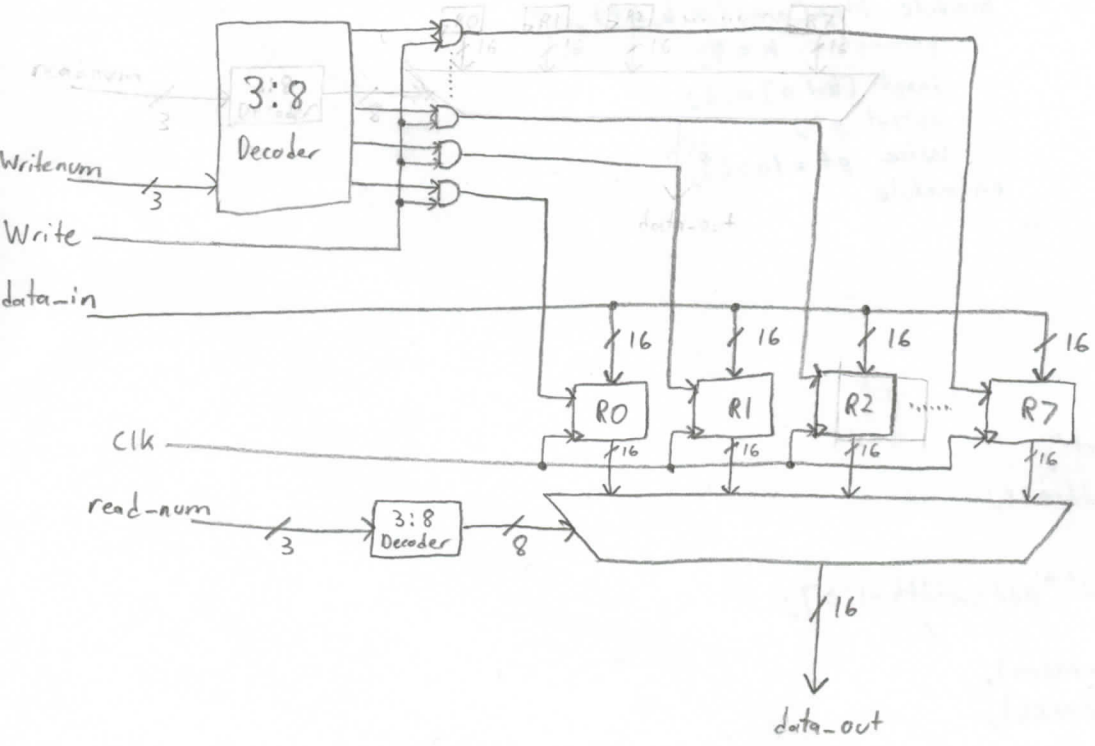



```
module vDFFE(clk, en, in, out);
  parameter n = 1; // width
  input clk, en;
  input [n-1:0] n;
  output [n-1:0] out
  reg [n-1:0] out
  wire [n-1:0] next_out;

  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++;`

Assembly: `LDR R5, [R2]` `LDR R6, [R2, #4]`

Machine language: `1100 0101` `1111 1000`

Control signals: ALUop[0:3] ← f(instrReg[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 powered off)

Reading from Memory

`LDR R5, [R2, #8]` (address)

// R2 contains <val> now, 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

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

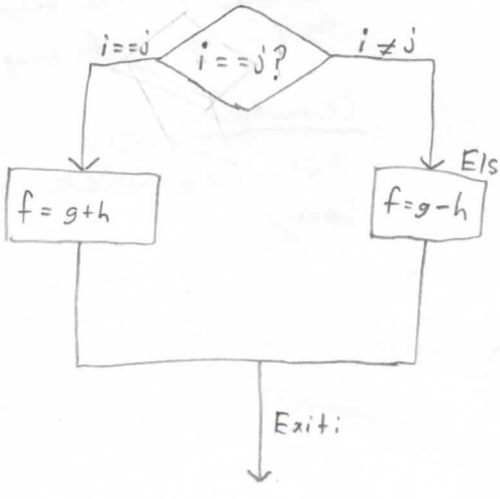
	Cond	F	I	Opcode	S	Rn	Rd	Operand2
# bits	4	2	1	4	1	4	4	12

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

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




```
CMP R3, R4
BNE Else // Branch not equal, so 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, r1, r2
  B  Exit
Else: SUB r0, r1, 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:

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

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

$10 \cdot 16^3 + 11 \cdot 16^2 + 0 \cdot 16 + 9 \cdot 1 = 43,785$

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
```

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;
}
```

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)

Terminology

Caller: makes the call ie: main

Callee: function being called ie: leaf_example

Convention:

- ARM uses Callee Saving