

ECE253 Abridged

Aman Bhargava

September 2019

Contents

1	Review: Bit Manipulation	4
1.1	Converting to and from Different Bases	4
1.1.1	Converting from base 10 \rightarrow base 2	4
1.1.2	Converting from base 2 \rightarrow base 16	4
1.1.3	Converting from base 10 \rightarrow base 16 (and vice versa) . .	5
2	Logic Functions and Logic Gates	6
2.1	Or Gate	6
2.2	And Gate	6
2.3	Inverter	6
2.4	XOR	6
3	Boolean Algebra	7
3.1	Useful Boolean Expression Rules	7
3.2	Sum-of-Products (SOP)	8
3.3	Product of Sums (POS)	8
3.4	NAND and NOR Logic Networks (TB 2.7)	9
3.5	Three-Way Light Control (TB 2.8.1)	9
4	How 2 Verilog	10
4.0.1	Code: 3-Way Multiplexer	10
4.1	Full Adder	11
4.2	7-Segment Display	14
4.2.1	Displaying Numbers	14
4.3	FPGA's	15
5	Karnaugh Maps	17
5.1	Motivation	17
5.2	Review of Terminology	17
5.3	Procedure for Minimum Cost Cover	18
5.4	5 Variable Karnaugh Map	18

6	Storage Elements	19
6.1	Introduction	19
6.2	RS Latches	19
6.2.1	Behavior	19
6.3	Gated RS Latch	20
6.3.1	Synchronous Reset	20
6.3.2	Behavior	20
6.4	Gated D Latch	20
6.4.1	Behavior:	20
6.5	D Flip-Flops	20
6.5.1	Master-Slave Flip Flop	21
6.5.2	Behavior	21
6.5.3	Why it's Useful	21
6.6	T Flip-Flop	21
6.7	Flip-Flop Reset/Preset	22
6.8	Summary of Objects	22
6.9	Verilog Implementations	22
6.9.1	Gated D-Latch	22
6.9.2	Edge-Triggered Flip Flop	22
6.9.3	Synchronous Reset (Active Low)	23
7	Finite State Machine	24
7.1	Review	24
7.1.1	State Diagrams	24
7.1.2	Lmao what is an always block actually	24
7.1.3	Case Statements	25
7.1.4	State Machine Diagrams and Sequential Diagrams	25
7.1.5	Case Statements for State Machines	25
8	Introduction to Microprocessors	27
8.1	Computer Organization	27
8.1.1	Memory Unit	27
8.1.2	Arithmetic Logic Unit (ALU)	28
8.1.3	Control Unit	28
9	Basic Concepts in Microprocessors	29
9.1	Basic Concepts	29
9.2	Memory Locations and Addresses	29
9.2.1	Memory Operations	30
9.3	Instruction Sequencing	30
9.3.1	RISC and CISC Instructions	30

9.4	Branching	31
9.5	Addressing Modes	31
9.6	More on Assembly	32
9.7	Stacks	32
9.8	Subroutines	32
9.8.1	Subroutine Nesting	33
9.8.2	Parameter Passing	33
9.8.3	Stack Frame	34
10	IO Devices	35
10.1	Program Control IO	35
10.2	Interrupts	35
10.2.1	Notes on SUBROUTINE INTERRUPT IMPLEMENTATION:	36
10.2.2	Enable and Disabling Interrupts	36
10.2.3	Process for a Single Device Interrupt Call	36
10.3	Multiple Device Interrupts	37
10.3.1	Approach One: Naïve Polling	37
10.3.2	Vectored Inputs	37
10.3.3	Interrupt Nesting	37
10.4	Processor Control Register	37

Chapter 1

Review: Bit Manipulation

Have you ever wanted to be a cool computer person who does things with ones and zero's instead of actual letters and numbers like a normal person? If so, this is the right chapter for you!

1.1 Converting to and from Different Bases

Base 10, 2, and 16 are most commonly used. Base 16 is just a way to read base 2 in a more efficient manner. In order to work with bits it's pretty important to know how to convert back and forth because the test is all on paper.

1.1.1 Converting from base 10 \rightarrow base 2

You keep dividing by two, keeping track of the remainder. Eventually the number you will be trying to divide by two will be 1. You keep going until it's zero + remainder(1). Then you read the remainders upward from that final 1.

1.1.2 Converting from base 2 \rightarrow base 16

Any hex number can be expressed as 4 binary digits. Make a correspondence table between quadruplets of binary numbers and hex (1-f, inclusive). To convert to base 16 subdivide from right to left in groups of four binary digits. Pad the leftmost part with leading zeros and convert using the table.

1.1.3 Converting from base 10 \rightarrow base 16 (and vice versa)

Just go through base 2 fam.

Chapter 2

Logic Functions and Logic Gates

2.1 Or Gate

1. Symbols
2. Switch structure
3. Truth table

2.2 And Gate

2.3 Inverter

2.4 XOR

Chapter 3

Boolean Algebra

Here are the axioms of Boolean Algebra:

1. $0 \cdot 0 = 0$
2. $1 \cdot 1 = 1$
3. $0 \cdot 1 = 1 \cdot 0 = 0$
4. if $x = 0$, $!x = 1$

Dual Form

1. $1 + 1 = 1$
2. $0 + 0 = 0$
3. $1 + 0 = 0 + 1 = 1$
4. if $x = 1$, $!x = 0$

Duality: In a given logic expression, you can swap $1 \rightarrow 0$ and $\cdot \rightarrow +$ and the expression is still valid.

3.1 Useful Boolean Expression Rules

- $x \cdot 0 = 0$
- $x \cdot 1 = x$
- $x \cdot x = x$

- $x \cdot !x = 0$
- $x \cdot 0 = 0$
- $!!x = x$
- $x + 1 = 1$
- $x + 0 = x$
- $x + x = x$
- $x + !x = 1$

Distributive Properties:

$$x \cdot (y + z) = xy + xz$$

$$x + (y \cdot z) = (x + y) \cdot (x + z)$$

3.2 Sum-of-Products (SOP)

'Sum' means boolean OR while 'product' means boolean AND.

Min term: for n variables, term where all variables appear once is a 'minterm'. Note that variables can either be complimented or uncomplimented.

Any boolean function can be represented by the sum of products of minterms - this is just done by simply converting the truth table and OR-ing each truth.

Then, you can simplify. Pretty common sense. 'Canonical' just means it's a bunch of midterms separated by OR's.

3.3 Product of Sums (POS)

Max term: where all n variables appear OR-d. They can be complimented or un-complimented.

You can pretty easily make all the valid max terms from a truth table. To generate a POS expression, you multiply (AND) the maxterms that sum to zero.

- 3.4 NAND and NOR Logic Networks (TB 2.7)**
- 3.5 Three-Way Light Control (TB 2.8.1)**

Chapter 4

How 2 Verilog

A 3-input multiplexer can be made with two 2-input multiplexers. Now let's implement this without using two pre-made multiplexers. Let:

-

Notes on Implementation

- If you want inputs to be registered from switches, you need to assign them `SW[jinti]`
- Likewise, if you want outputs to be registered to LED's, assign them `LEDR[jinti]`

4.0.1 Code: 3-Way Multiplexer

```
module mux2b2to1 (SW, LEDR); //Two bit 2 to 1 multiplexer
    input[4:0] SW, //[4:0] sets switches 0-4 to inputs(?)
    output[1:0] LEDR, //

    wire S;
    wire[1:0] a, b, z; //'two bit wide vector'?

    assign a = SW[1:0],
    assign b = SW[3:2],
    assign s = SW[4],
    assign LEDR = z,

    assign z[0] = (~s&a[0]) | (s&b[0]);
    assign z[1] = (~s & a[1]) | (s&b[1]);
```

```
endmodule;
```

Can you make the assignment more efficient? What if you do this:

```
// assign z[0] = (~s&a[0]) | (s&b[0]);  
// assign z[1] = (~s & a[1]) | (s&b[1]);  
assign z = (~s&a) | (s&b); // NOT CORRECT
```

Because s is only 1-bit and a, b are two bits, s is extended with a 0, which makes the logic incorrect!

General Notes on how Syntax Works:

- **What is 'assign'?** 'assign' just means you're making a connection (alias?) b/w the two. **Question:** is this necessary for instantiating the variable?
- **Assignment arith.** When you create a verilog wire/input with $x = [a : b]$, the number of bits in x is $b - a + 1$
- **Bit Access:** To access bits, you say $x[n]$ or $x[n : m]$ where $n < m$. The length of the slice is $n - m + 1$
- **Concatenation:** If you want to stitch together multiple bits, you use $x = SW[9 : 8], SW[1 : 0]$. That statement stitches together two 2-bit chunks ($SW[9 : 8], SW[1 : 0]$ to make one 4-bit chunk).
- **Order of Operations:** Basically nobody knows... just use parenthesis when you're not sure. And goes before or, though.

4.1 Full Adder

Description: Adding in binary is the same as in decimal, just with fewer options. When adding two single bits, we have three output possibilities: 00, 01, 10. We call the least significant bit the sum s and the most significant digit the carry c .

	x	y	C	s
	0	0	0	0
Let's see the truth table:	0	1	0	1
	1	0	0	1
	1	1	1	0

As you can see, the sum bit is just $x \oplus y$ and the carry bit is xy . This is a **half adder** because it doesn't accept a carry from the last calculation.

A full adder accepts a carry in C_{in} , x , and y , and outputs sum s and carry out C_{out} .

	x	y	C_{in}	C_{out}	s
	0	0	0	0	0
	0	0	1	0	1
	0	1	0	0	1
Let's see the truth table:	0	1	1	1	0
	1	0	0	0	1
	1	0	1	1	0
	1	1	0	1	0
	1	1	1	1	1

Therefore

$$C_{out} = xy + xC_{in} + yC_{in}$$

and

$$s = x \oplus y \oplus C_{in}$$

Ripple Carry Adder: Then you can string them together by assigning one of these adders to each bit of the output and passing the carry out to the carry in of the next bit!

Specifics for Ripple Carry Adder:

- Max size for output is one more than the two input's sizes because one more order of magnitude in binary is just doubling the number, and the most you can do when adding two numbers of equal length is double them.
- Basically just string them together and you're golden.

Review of Outcomes

- $C_{out} = xy + C_{in}X + C_{in}Y$
- $Sum = x \oplus y \oplus z$

Now let's make it in verilog!

```
module fulladder(x, y, Cin, S, Cout);
    input x, y, Cin;
    output S, Cout;

    assign s = x ^ y ^ Cin;
    assign Cout = (x&y) | (Cin&x) | (Cin&y); // Why no wires here?
        Bc no physical IO's.
```

Now we make a 3-bit adder out of full adders in Verilog!

```
module adder3bit(X, Y, S)
    input[2:0] X, Y;
    output[3:0] S;
    wire[3:0] C; // to connect full adders together

    fulladder U0(X[0], Y[0], C[0], C[1]);
    fulladder U1(X[1], Y[1], C[1], C[2]);
    fulladder U2(X[2], Y[2], C[2], C[3]);

    assign S[3] = C[3]; // Final carry bit is the most significant
        bit of the sum.
    assign C[0] = 1'b0; // Weird syntax: 1 bit, equal to 0.
endmodule;
```

This is structural verilog - we can't immediately see the bigger picture. We see wires and modules stitched together and we have to figure out what it all means.

Weird constant syntax:

- `1'b0`: 1 bit constant, in binary, equal to 0.
- `4'hF`: 4 bit constant, in hex, equal to F.
- `4'd9`: 4 bit constant, in decimal, equal to 9.
- `8'h1E`: 8 bit constant, in hex, equal to 1E.

4.2 7-Segment Display

Prompt: Design a circuit with two inputs x_1 and x_0 representing a 2-bit number x . Show x on the 7-segment display (ranges from 0-3 inclusive).

Numbering: h_0 is the top segment. Clockwise increases. h_6 is the middle one.

Note on D1-SoC Board: Logic 0 makes the light turn on for h_{0-6} and 1 makes it turn off ('active low')

4.2.1 Displaying Numbers

Sections to light up for 0: h_{0-5} Sections to light up for 1: h_{1-2} Sections to light up for 2: h_{0-1}, h_6, h_{3-4} Sections to light up for 3: h_{0-3}, h_6

With active low: Sections to power for 0: h_6 Sections to power for 1: h_0, h_{3-6} Sections to power for 2: . Sections to power for 3:

	x_1	x_2	h_6	h_5	h_4	h_3	h_2	h_1	h_0
	0	0	1	0	0	0	0	0	0
Truth Table:	0	1	1	1	1	1	0	0	1
	1	0	0	1	0	0	1	0	0
	1	1	0	1	1	0	0	0	0

Consolidating the Logic Functions:

$$h_0 = (!x_1) \cdot x_0$$

$$h_1 = 0$$

$$h_2 = x_1 \cdot !x_0$$

$$h_3 = !x_1 \cdot x_0$$

$$h_4 = x_0$$

$$h_5 = x_1 | x_0$$

$$h_6 = !x_1$$

Verilog Code:

```
module seg7(SW, HEX0)
  input[1:0] X;
  output[6:0] HEX0;

  assign HEX0[0] = ~SW[1] & SW[0];
  // etc.

endmodule
```

Implementation Example: Design a circuit with 4 inputs a, b, c, d and s . if $s = 0$, show $a + b$ on 7-seg display if $s = 1$, show $c + d$ on 7-seg display

$$a, b \rightarrow mux_{2bit*2inputs} \rightarrow fulladd_{cin=0} \rightarrow HEXO \rightarrow output$$

- The carry in digit to the full adder is 0
- The carry out from the full adder is the most significant digit of the HEXO input.
- Honestly you could just use a half adder because you don't need a carry-in. $C_{out} = xy$, $Sum = x \oplus y$

Cool XOR Fact: \oplus returns 1 if the number of true inputs is ODD.

4.3 FPGA's

Can do anything.

- Field programmable gate arrays: Programmable in the field (i.e. not in the factory necessarily).
- 2-D array of programmable blocks.
- Blocks contain lookup tables (LUT)
- Any two input LUT can be programmed for any two-input logic functions.
- More complex logic comes out when you connect the LUTS - this is what Quartus does.

How to encode any two-input thing Let there be 4 sRAM bits p, q, r, s .
Let the inputs be a, b .

$$p, q \rightarrow mux_{a1} \rightarrow x$$

$$r, s \rightarrow mux_{a1} \rightarrow y$$

Where x, y are intermediate variables.

$$x, y \rightarrow mux_b \rightarrow output$$

Therefore it takes 4 ram cells for a 2-input LUT.

$$n_{ramcells} = 2^{inputs}$$

There are 85,000 6-LUTS on lab chips.

Chapter 5

Karnaugh Maps

5.1 Motivation

Creating boolean functions to get what you need done is confusing and unlikely to lead to an optimal solution when there are many variables.

Karnaugh maps are a tool for expressing your truth tables in such a way that makes getting optimal solutions easier with many variables.

Example 2x2 Karnaugh Map		0	1
	0	a	b
	1	c	d

5.2 Review of Terminology

- **Implicant:** Any product term for which the function is true (think 'implies' logic is 1)
- **Cover:** A set of implicants that covers all 1's of the function.
- **Prime implicant:** Any implicant that, if we delete a literal, is no longer an implicant (largest groups of ones in the K-map)
- **Essential prime implicant:** Prime implicant that covers a 1 covered by no other prime implicant.
- **Min-cost cover:**
- **Cost:**

Example:

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

Let the left column be the z column and the top be products of x, y .

List of implicants:

1. \bar{z}
2. $\bar{x}\bar{y}$
3. $\bar{x}\bar{z}$
4. $y\bar{z}$
5. $x\bar{z}$
6. $\bar{y}\bar{z}$
7. $m_{0,1,2,4,5}$

Prime Implicants: Largest group of ones in cardinal maps...

Cost Definition: $n_{gates} + n_{inputs}$

5.3 Procedure for Minimum Cost Cover

- Find essential prime implicants and include in cover
- Select additional prime implicants to include in the cover until all 1's are covered.

5.4 5 Variable Karnaugh Map

Use two 4-variable K-Maps. One for case where $x_5 = 0$, other for $x_5 = 1$. Min-term indexing goes by the binary representation of x_{0-n} where x_n is the LEAST significant bit.

Chapter 6

Storage Elements

6.1 Introduction

Until now, the values coming in have depended on the state (high or low) of the input wires to the circuit. There's another option where the inputs depend on past states of the circuit.

Storage elements represent a **state** of a circuit. When inputs change, new inputs either leave circuit in the same state or change the state. These circuits are called **sequential** circuits. Memory elements can be created with logic gates.

6.2 RS Latches

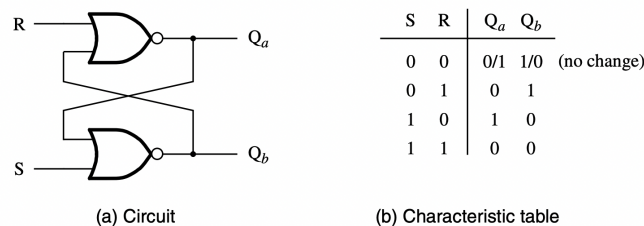


Figure 6.1: Basic Latch

6.2.1 Behavior

You use S to set the output Q_a to 1 and R to set Q_a to 0.

S	R	Q_a	Q_b
0	0	No change	
0	1	0	1
1	0	1	0
1	1	0	0

S functions as the 'set' signal.

6.3 Gated RS Latch

Add an and gate to the inputs so that you can only enter signals to the latch when the clk signal is 1.

6.3.1 Synchronous Reset

This is when you can only use the reset when the clock is at 1 (basically you just AND the D and the R_n for an 'active low' reset). Reset behavior is synchronized to the rising edge.

6.3.2 Behavior

Same as an RS latch, but you can only send R and S signals when the clk is logic 1.

6.4 Gated D Latch

Also called 'Transparent Latch' or 'Level-Sensitive Latch'.

6.4.1 Behavior:

When the clock is 1, then $Q = D$. Q changes as D changes and then when the clock is turned back to 0, then Q persists in it's last state.

6.5 D Flip-Flops

Symbol: Same as the D Latch, but there's a triangle instead of clk . If it's a **negative edge** triggered flip-flop, the symbol has a bubble next to the clock. **Positive edge** appears to be the default.

Edge-Triggered Flip Flop: The flip flop only changes when the clock is on a rising edge or a falling edge.

6.5.1 Master-Slave Flip Flop

You connect a master D-Latch to a slave D-Latch. The clock signals of one is the inverse of the clock symbol going into the other one.

This setup yields a system where you can only change the value of Q for the slave (Q_s) on a rising/falling edge.

6.5.2 Behavior

On rising or falling clock edge, the value of D is stored and appears in Q .

Work-Through demonstration: Let's say you have the inverted clock going to the slave. When you're in logic 1 for the clock, the master is in **transparent** mode. $Q_m = D$. When you switch, the value of the master is stuck and the value of the slave is set to Q_m . You can't reset the slave value after the transition because the **opaque** mode of the master.

Pretty cool property!

6.5.3 Why it's Useful

You can put these in the middle of circuits to save values. Then, you can re-inject inputs while you wait for the last half of the circuit to finish processing. It's like splitting the circuit into two pieces that can run on different elements simultaneously.

This is a bit of a dumb way to think about it, but that's the general idea behind how they are actually used.

6.6 T Flip-Flop

Let's take a D Flip Flop component and loop the value of Q and \bar{Q} back to the input D . Let's make T the new input. T switches the input of D between Q and \bar{Q} .

6.7 Flip-Flop Reset/Preset

6.8 Summary of Objects

6.9 Verilog Implementations

6.9.1 Gated D-Latch

```
module D_Latch (D, clock, Q, Qb);
    input D, clock;
    output reg Q, Qb; // 'reg' means it's an always block

    always@(D,clock) //defining a block that's sensitive to changes
        in Q and D.
    begin
        if(clock==1'b1)
            begin // need this begin because you're doing two things in
                the if block.
                    Q = D;
                    Qb = ~D;
            end // begin and end function like curly braces in C/C++/Java
    end //if clock == 0'b1, then verilog maintains previous values.

endmodule
```

6.9.2 Edge-Triggered Flip Flop

```
module D_ft(D, clock, Q, Qb)
    input D, clock;
    output reg Q, Qb;

    // Always blocks tell you what you need to
    always@(posedge clock) // This means you execute this on the
        positive edge (posedge is a keyword for that)
    begin
        Q <= D; // use '<=' for describing flip flops.
        Qb <= ~D; // we may get to why in this course but for now
            it's just a thing...
    end
endmodule
```

6.9.3 Synchronous Reset (Active Low)

```
always@(posedge clock)
begin
    if(resetn == 1'b0)
    begin
        Q <= 1'b0;
        Qb <= 1'b1;
    end
    else
    begin
        Q <= D;
        Qb <= ~D;
    end
end

endmodule;
```

Chapter 7

Finite State Machine

7.1 Review

A finite state machine (FSM) is an object that changes its properties (outputs) based on inputs over time. For example, a human could be modeled as a state machine where the input is water over time and the output is the need to go to the bathroom. **Sequential circuits** are FSM's.

7.1.1 State Diagrams

Example: w can be 0, 1. We are trying to find when w has gone $1 \rightarrow 0 \rightarrow 1$.

Solution: We can use a shift register to get all the sequential bits of w in, then we use a simple logical expression on the outputs $w_0\dot{w}_1\dot{w}_2$.

One-Hot Encoding: When only one of your bits is one. **one cold** is the opposite.

7.1.2 Lmao what is an always block actually

```
always@(A, B, C)
begin

end
```

A, B, C are in the sensitivity list, the code inside is ‘sensitive’ to them so as they change the internal block is fired. You can only use **if** statements in here.

7.1.3 Case Statements

Same as in pretty much all other programming languages.

```
case(A)
  value1:
    do_something;
  value2:
    do_something_else;
  value3: // multiple lines
    begin
      do_something;
      do_something more;
    end
  default:
    default_behavior;
endcase
```

7.1.4 State Machine Diagrams and Sequential Diagrams

7.1.5 Case Statements for State Machines

Quick example:

```
module FSM(input w, clock, resetn, output z);
  reg[1:0] y, Y;
  parameter A = 2'b0, B = 2'b01, C = 2'b10, D = 2'b11;
  //state table
  always@(w, y) begin
    case(y) begin
      A: begin
        if(W)
          Y = B;
        else
          Y = A;
        end
      B: begin
        if(w)
          Y = B;
        end
    end
  end
```

```

        else
            Y = C;
        end
    endcase
end

always@(posedge clk) begin
    if(!resetn)
        y <= A;
    else
        y <= Y;
    end

    assign z = (y == D);

endmodule

```

- *parameter* lets you define constants.

Chapter 8

Introduction to Microprocessors

8.1 Computer Organization

A **computer** can read, write, and process data. In order to do so, it has the following main parts:

1. Input/Output Devices
2. Memory
3. Processor (includes Arithmetic Logic Unit and Control Unit)

All of these are connected through an **interconnection network**.

8.1.1 Memory Unit

The memory unit can be broken into three main pieces. Overall, it functions to **store programs** and **data**.

Primary Memory

- Stores programs
- Relatively fast, called ‘main memory’
- Made of a collection of **bit registers**

The bit registers in memory are read and written in **WORDS**. Each word has its own address (consecutive numbers) and a standardized number of bits.

Cache

- Smaller, faster RAM unit (RAM = Random Access Memory, all can be accessed at the same speed).
- Holds current parts of a program.
- Packaged with the processor → faster to access information from here than from memory.

Secondary Storage

Basically just external drives.

8.1.2 Arithmetic Logic Unit (ALU)

Most operations are executed here (+, −, /, ·, ≤, ≥, >, <). Operands are passed here via **registers**. Registers are 1 word long and are very fast to access.

8.1.3 Control Unit

Coordinates the rest of the units. Usually distributed throughout the computer. The following main types of operations are coordinated by this unit:

1. Storing information in memory as programs and data.
2. Moving information to memory for the ALU to process.
3. Moving processed information to the output unit.

Chapter 9

Basic Concepts in Microprocessors

9.1 Basic Concepts

LOAD	R2, LOC	// Copies data from main memory into a register.
ADD	R4, R2, R3	// R4 = R2 + R3 (contents)
STORE	R4, LOC	// Copies data from a REGISTER to main memory.
MOVE	R2, R4	// Copy contents of R4 to R2
CLEAR	R2	// Set contents of R2 to be 0
ADD	R2, R2, R4	// R2 = R2 + R4
SUB	// Same as add but subtraction	
MULTIPLY		
DIVIDE		
AND	R4, R2, R3	// R4 = R2 & R3
OR		
NOT		

The processor's **PROGRAM COUNTER** (PC) holds the address of the next instruction. Each instruction is 1 word in RISC computers.

9.2 Memory Locations and Addresses

1 cell is 1 bit, and n cells is 1 word. Memory is a collection of words, usually from 16-64.

Address spaces: The possible addresses in a k -bit address space range from $0 \rightarrow 2^k - 1$

Byte-addressable memory indexes each byte (8 bits) with an integer $0, 1, 2, \dots, n$. Therefore, in a 32-bit word system, the address of each word is $0, 4, 8, 12, \dots$ in byte addressable memory.

Big endian means that the lower byte address has the more significant digit while **little endian** holds the opposite.

9.2.1 Memory Operations

READ and WRITE are the main operations.

- **READING** copies the contents of a memory location to a processor register. This is done by specifying the address in memory that the processor wants to read.
- **WRITING** involves copying the contents of a processor register to a particular place in memory. The process sends the address it wishes to write to and the data it wants to write there.

9.3 Instruction Sequencing

There are 4 operations needed for running a computer program:

1. Data transfers between memory and process (READ and WRITE)
2. Arithmetic and logic operators
3. Program sequencing and control (branching, subroutines, etc.)
4. Input and Output

9.3.1 RISC and CISC Instructions

RISC: Each instruction is 1 word long. All arguments must be in registers already.

CISC: Each instruction can be ≥ 1 word long. This enables more complex operations.

9.4 Branching

So far we have talked about **straight line sequencing**. Instructions are executed in the exact order they are written. But what if we want to loop?

```
// Some computation
BGT    R4, R5, LOOP    // Go to LOOP if R4 > R5
// Stuff to be executed if R4 <= R5...

LOOP:
    // To be executed if R4 > R5

// SIDE NOTE: This is how you use constants:
SUBTRACT R2, R2, #1 // R2 = R2 - 1
```

9.5 Addressing Modes

A register holding the address of some information in memory is a **pointer**. Using pointers is called the *indirect addressing mode*.

```
LOAD    R2, R5          // Loads the stuff from the MEMORY ADDRESS
                        that R5 holds into R2.
MOVE     R2, R5          // Puts the IMMEDIATE VALUE held in R5 into
                        R2.
MOVE     R2, #300        // R2 = 300;
CLEAR    R2              // CLEARS R2 to 0's.

ADD      R4, R0, #200    // R0 usually holds 0's. This what MOVE
                        really does.
```

Index mode of addressing is when you use an address register to hold the address of the first word in a list.

```
// Memory address is equal to some value X (let's say address #96
    in memory) plus a register value at Ri
X(Ri) // This is how you address that memory address.
```

9.6 More on Assembly

```
TWENTY EQU 20 // Whenever the assembler sees 'TWENTY' it will be
               replaced by 20.

ORIGIN 100     // Insert this at the beginning of a block to
               specify that code below should be placed in memory starting at
               address #100

SUM   : Reserve 4 // 4-byte space reserved @ address 100
N     : Dataword  // N = #150 @ address 204
End // ends the program.

%10110101 // how to specify numbers in binary
0x5f      // how to specify number in hex
```

9.7 Stacks

Same common datastructure we ran into before in CSC190. Only one end can be read/added to, and it's LIFO (last in, first out).

The Stack Pointer (SP) points to the address of the end of the stack (Processor stack). Common practice: stack grows in direction of *decreasing memory addresses*. The push operation works as follows (pushing word in R_i):

```
SUB      SP, SP, #4 // Decrementing stack pointer by 4 bytes
           (1 word).
STORE    Ri, SP
// Now the pop operator:
LOAD     Ri, SP
ADD      SP, SP, #4
```

9.8 Subroutines

Same as a function. Uses branching, but it knows that it has to return to where it was called from. We use the **RETURN** command to return to the calling place. The address of the calling place is stored in the **LINK REGISTER**.

```

CALL      FUNCNAME      // Stores current PROGRAM COUNTER in LINK
                        REGISTER, branches to subroutine.
// .. some code ..
FUNCNAME:
    // .. some code ..
    RETURN              // Branches back to address stored in LINK
                        REGISTER

```

9.8.1 Subroutine Nesting

If you call a function in a function, the LINK REGISTER can only hold one address at a time. Since we need LIFO structure to tell us where to branch back to at each successive return, we use the **PROCESSOR STACK** to store the further addresses.

Process for calling a nested function:

1. Return address in LR \rightarrow PROCESSOR STACK @SP.
2. Store current address in LR and branch to the next function.
3. When you return from the nested function, transfer the original LR pointer back from PROCESSOR STACK @SP.
4. Return as normal to address in LR when you're done with the original subroutine.

9.8.2 Parameter Passing

This can be done with registers or via the **PROCESSOR STACK**. Here are the steps for the PROCESSOR STACK approach:

1. Push the arguments you want to pass to the stack in the caller.
2. Once in the subroutine, save the contents of current registers to the stack to preserve the state of the calling program.
3. Use an offset equal to the number of SAVED REGISTERS to access passed variables.
4. Carry out the subroutine.

5. Restore the disturbed registers from the stack (pop them off).
6. Branch back to the address of the caller and re-establish the SP to what it was originally.

9.8.3 Stack Frame

Definition: The Stack Frame is the space allocated to a subroutine when it is called. It is useful to have a **FRAME POINTER** that points to just above the PASSED PARAMETERS (unlike the STACK POINTER SP that points to the actual top of the stack that also holds the saved registers).

FRAME POINTER FP is a constant for the duration of the subroutine program. It is equal to the STACK POINTER SP at the beginning of the subroutine call.

Chapter 10

IO Devices

You access IO devices in the same way you might access regular registers. The **device interface** gives you a few registers that allow you to communicate:

1. Data
2. Status
3. Control

10.1 Program Control IO

You use a program called a **Program Control Interface** to write and control an IO device.

Because the processor, input, and output devices may work at different rates, you need to wait for a response to any signal you send to one to make sure it's ready for the next one. These are called **STATUS FLAGS**, and reading them is called **POLLING**.

10.2 Interrupts

You can use the infinite loop approach where you just keep POLLING the STATUS FLAG of an IO device until it's ready. That takes a lot of compute power and you can't do anything during the wait time, which is a problem.

INTERRUPT SERVICE ROUTINE is called by an interrupt signal from an IO device. Here is the processor's steps for dealing with one.

1. The processor finishes instruction i that it is in the middle of.

2. The processor branches to the INTERRUPT ROUTINE. PC \rightarrow PROCESSOR STACK.
3. The INTERRUPT ROUTINE is executed.
4. The PC is restored from the PROCESSOR STACK.

10.2.1 Notes on SUBROUTINE INTERRUPT IMPLEMENTATION

At the beginning of the subroutine, the processor must send an **INTERRUPT ACKNOWLEDGE** make sure that, when the subroutine RETURNS, an infinite loop does not occur. Alternatively, the successful execution of the subroutine can implicitly notify the IO device.

REGISTERS MUST BE RESTORED because you don't know when the interrupt subroutine will be executed. It could be in the middle of another very sensitive subroutine.

10.2.2 Enable and Disabling Interrupts

Sometimes you don't want to branch away from the main program. Therefore, the processor has some STATUS REGISTERS PS that have a bit INTERRUPT INABLE IE that, when set to 0, disable interrupts.

You must set this to 0 while inside the subroutine to avoid an infinite loop..

IO MODE bit in the control interface of the IO device can also allow you to set whether it sends interrupts or not.

10.2.3 Process for a Single Device Interrupt Call

1. Device raises interrupt
2. Processor branches to the interrupt subroutine
3. STATUS REGISTER PS is saved to the stack, along with the PROGRAM COUNTER PC.
4. STATUS REGISTER PS has INTERRUPT ENABLE IE that is set to 0.
5. Subroutine runs.
6. STATUS REGISTER PS is restored from the saved version.

10.3 Multiple Device Interrupts

The above system works fine when you just have one interrupt at a time or one device performing the interrupts.

10.3.1 Approach One: Naïve Polling

You can just do the same thing as before, just executing the interrupts in the order you poll them if they happen to come at the same time. However, polling takes time.

10.3.2 Vectored Inputs

If you allow the interruptor to send a message along with their interrupt, you can have code in the processor that tells it how to interpret the signal.

INTERRUPT VECTOR TABLE is stored (usually in the lowest address range). The information in a vectored interrupt contains the address of one of the vectors in that table. The contents of that vector in the table is the address of the interrupt subroutine.

10.3.3 Interrupt Nesting

If an interrupt comes in while executing another interrupt's subroutine, there's a chance that it will be of higher priority. Therefore, we need an interrupt priority system.

PROCESSOR PRIORITY is the priority of the current process. Only interrupts with a higher priority are executed. Priority is communicated in the interrupt vector.

10.4 Processor Control Register

We already know that the INTERRUPT ENABLE IE bit is in the PROCESSOR STATUS register which is one of the PROCESSOR CONTROL REGISTERS.

IPS is where PS is saved automatically at the start of an interrupt. PS is autorestored from here at the end.

IENABLE register enables device specific INTERRUPT ENABLE bits.

IPENDING register shows which interrupts are active.

Since all registers are 1 word long, you can generally have 32 IO devices.

In order to read and write from PROCESSOR CONTROL REGISTERS, you treat them like regular registers.