ARM Instructions

- Five general formats of ARM instructions
- Format refers to how many and what type of operands
  - Registers are used like a variable in an ARM instruction, or
  - Data Memory:
    - ★ data memory accesses are always multiples of 8

Remember It

Instruction	**Format**	Example	Meaning
add	**R-format**	`ADD X1,X2,X3`	`X1 = X2 + X3`
subtract	**R-format**	`SUB X1,X2,X3`	`X1 = X2 - X3`
addi	**I-format**	`ADDI X1,X2,#C`	`X1 = X2 + C`
subi	**I-format**	`SUBI X1,X2,#C`	`X1 = X2 - C`
load word	**D-format**	`LDUR X1,[X2,#Imm]`	`X1` = Memory[`X2+Imm`]
store word	**D-format**	`STUR X1,[X2,#Imm]`	Memory[`X2+Imm`] = `X1`
branch	**B-format**	`B #Imm`	`PC = PC + 4` × `Imm`
branch on zero	**CB-format**	`CBZ X1,#Imm`	if (`X1==0`) PC = PC + 4 × Imm else PC = PC + 4
branch on non-zero	**CB-format**	`CBNZ X1,#Imm`	if (`X1!=0`) PC = PC + 4 × Imm else PC = PC + 4

- `#Imm` are signed constants, and `#C` are unsigned constants

Range of Values of #Imm and #C

Remember It

Instruction	Signed/Unsigned	Inclusive Range
ADDI/SUBI	Unsigned	$[0, 4095]$
LDUR/STUR	Signed	$[-256, 255]$
B	Signed	$[-33,554,432$ to $33,554,431]$
CBNZ/CBZ	Signed	$[-262,144$ to $262,143]$

Example 1 - Writing ARM Code

Try this

Write ARM code that accomplishes the following high-level language objective:

```
a = b + c - d;
```

Assume, the values of the variables `a`, `b`, `c`, and `d` are in registers `X0`, `X1`, `X2` and `X3`, respectively.

Try this

Write ARM code that accomplishes the following high-level language objective:

```
B[5] = B[4]
```

Assume, the byte address of `B[0]` is in general purpose register X1.

Example 2 - Writing ARM Code

Try this

Write ARM code that accomplishes the following high-level language objective:

```
a = a + 7
b = b - 1
```

Assume, the values of the variables `a` and `b` are stored in registers `X1` and `X2`, respectively.

Think About It

- Why have both ADDI and SUBI instructions?
  Is `ADDI X1, X2, #-10` $\equiv$ `SUBI X1, X2, #10`?
- `immediate` **cannot** be negative in I-Format instructions

Example 3 - Writing ARM Code

Try this

What are the first five values of PC when the following code is executed, with PC=304:

```
PC →   304:   B #3
       308:   ADD X1, X2, X3
       312:   SUB X1, X3, X5
       316:   ADDI X2, X12,#16
       320:   B #-2
```

Try this

If PC=100, how many iterations of the loop are executed?

```
PC →   100:   ADD X1, XZR, XZR
       104:   ADDI X2, XZR, #6
       108:   ADDI X1, X1, #5
       112:   SUBI X2, X2, #1
       116:   CBNZ X2, #-2
       120:   ADD X4, X6, X8
```

Performance

- How can we compare different computer designs?
- Two important performance metrics:
  - Response time: time between start and completion of a task
  - Throughput: total number of tasks completed in a given unit of time
- Improving response time usually improves throughput
- How can we measure time?
- Nearly all computers have a clock.
- Clock cycles or ticks: discrete time when hardware events take place.
- **Clock period**: the length of a clock cycle
- **Clock rate** $= \frac{1}{\textbf{clock period}}$

Performance Metrics and Analysis

To improve performance:

- Reduce the number of clock cycles for a program
- Reduce the length of the clock cycle
- Replace a slow processor with a faster one.
  - Response time is faster, and so throughput will also increase
- Adding more processors, where each processor does **only** one task.
  - If all tasks are exactly the same, then only throughput increases.

Some factors affecting response time

- Speed of clock
- Complexity of instruction set
- Efficiency of compilers
- Mix of instructions needed to complete a task
- Some choices in designing computers:
  - simple instruction set, fast clock, one instruction executed per clock cycle
  - complex instruction set, faster clock, some instructions take multiple cycles to execute

Example: Small code changes, big performance differences

```
#include<stdio.h>
#define NR 10000
#define NC 10000

int a[NR][NC];

void main() {
  int i,j;
  for (i=0;i<NR;i++){
    for (j=0;j<NC;j++){
      a[i][j]=32767; } } }
```

- Row-by-row (a[i][j]): 1.693 sec

```
#include<stdio.h>
#define NR 10000
#define NC 10000

int a[NR][NC];

void main() {
  int i,j;
  for (i=0;i<NR;i++){
    for (j=0;j<NC;j++){
      a[j][i]=32767; } } }
```

- Down a column (a[j][i]): 27.045 sec (approx 16 times slower!)

Example Revisited: Memory access vs. Registers

```
#include<stdio.h>
#define NR 10000
#define NC 10000

int a[NR][NC];

void main() {
  register int i,j;
  for (i=0;i<NR;i++){
    for (j=0;j<NC;j++){
      ; } } }
```

- `register int i,j`: 0.044 sec

```
#include<stdio.h>
#define NR 10000
#define NC 10000

int a[NR][NC];

void main() {
  int i,j;
  for (i=0;i<NR;i++){
    for (j=0;j<NC;j++){
      ; } } }
```

- `int i,j`: 0.27 sec
  (approx 6 times slower!)

Benchmarks

- Standard set of programs (and data) chosen to measure performance
- Advantages:
  - Provides basis for meaningful comparisons
  - Design by committee may eliminate vendor bias
- Disadvantages:
  - Vendors can optimize for benchmark performance
  - Possible mismatch between benchmark and user needs
  - Still an artificial measurement

Conclusion

Lecture Summary

- ARM instruction formats and their use
- Read ARM code
- Write ARM code
- Performance metrics and measurements

Assigned Textbook Readings

- ARM Overview
- Section 1.6, just until top of page 30.
- **Skim** rest of the section

Next Steps

- **Read** assignment A1.
  - Start thinking about the questions in A1
- **Attempt** questions that are similar to the problem we solved in the lecture.
- **Ask** questions in the next tutorial or office hours.

Additional Slides

Remaining slides are additional notes for your information.

CS 251 vs CS 241

- CS 241 uses MIPS, CS 251 uses ARM.
- CS241 from HLL to Assembly
- CS251 Assembly down to hardware
- MIPS and ARM assembly similar:

MIPS	**ARM**
`lw $1, 0($2)`	`LDUR X1, [X2,#0]`
`add $1,$2,$3`	`ADD X1,X2,X3`
`addi $1,$2,22`	`ADDI X1,X2,#22`

- ARMv7 has 15 registers; ARMv8 (LEG) and MIPS have 32
  ARMv8,LEG: X31 is always 0
  MIPS: $0 is always 0
- ARM takes about 1/4 the number of transistors as MIPS
- ARMv7 has conditional forms of instructions
  ARM compiler takes advantage of this; gcc does not
  ARMv8 has fewer (than ARMv7) fancy features; we use none of them

ARM Register Format (R-Format) instructions

- Perform arithmetic or logic operations on data (operands) in registers
- Examples:
  - `ADD X1,X2,X3` $\equiv X1 = X2 + X3$
    Adds contents of X2 with the contents of X3; store result in X1.
  - `SUB X1,X2,X3` $\equiv X1 = X2 - X3$
    Subtracts contents of X3 from the contents of X2; store result in X1

ARM Data Format (D-Format) Instructions

- Move data between memory and registers
- Examples:
  - `LDUR X1, [X2, #24]` $\equiv$ X1 $\leftarrow$ MEM[X2 + 24]
    Load[2] data stored in memory byte address X2+24 into register X1.
  - `STUR X1, [X2, #32]` $\equiv$ X1 $\rightarrow$ MEM[X2 + 24]
    Store[3] contents of register X1 into memory at byte address X2+32
- Ensure data memory byte address is a multiple of 8

[2] LoaD Unscaled Register (LDUR)
[3] STore Unscaled Register (STUR)

ARM Immediate Format (I-Format) Instructions

- One operand is in a register, one operand is a constant (an `immediate`) embedded in the instruction
- Examples:
  - `ADDI X1, X2, #100` $\equiv X1 = X2 + 100$
    Adds constant 100 to contents of X2; store result in X1
  - `SUBI X1, X2, #10` $\equiv X1 = X2 - 10$
    Subtract constant 10 from the contents of X2; store result in X1

Think About It

- Why have both ADDI and SUBI instructions?
  Is `ADDI X1, X2, #-10` $\equiv$ `SUBI X1, X2, #10`?
- `immediate` **cannot** be negative in I-Format instructions

ARM Branch Format (B-Format) Instructions for Control Flow

- an **unconditional** goto statement is used to change the control flow in the execution of the instructions.
- Example: `B #28` $\equiv$ PC = PC + 4 $\times$ 28
- The `immediate` specifies a word offset relative to **current** PC

Think About It

- Why multiply by 4?
  That is the relationship between a word and a byte
- Can `immediate` be negative in control flow instructions?

ARM Conditional Branch Format (CB-Format) Instructions

- a **conditional** goto statement is used to change the flow in the execution of the instructions.
- Compare the contents of a general purpose register to **ZERO**
- Examples:

`CBZ X1,#8` is equivalent to

```
if X1 == 0 then
     PC = PC + 4 * 8
else
     PC = PC + 4
```

`CBNZ X1,#8` is equivalent to

```
if X1 != 0 then
     PC = PC + 4 * 8
else
     PC = PC + 4
```

- The `immediate` specifies a word offset relative to **current** PC

Disclaimer

The slides and lecture notes presented here are a combination of the CS251 course notes from previous terms, the work of Xiao-Bo Li, and material from the required textbook "Computer Organization and Design, ARM Edition," by David A. Patterson and John L. Hennessy. It is being used here with explicit permission from the authors.

CS251 course policy requires students to delete all course files after the term. Therefore, please do not post these slides to any website or share them.

CS251 - Computer Organization and Design

Intro to Digital Logic Design - Combinational Logic Design

Instructor: Zille Huma Kamal

University of Waterloo

Fall 2024

A Brief Look at Electricity

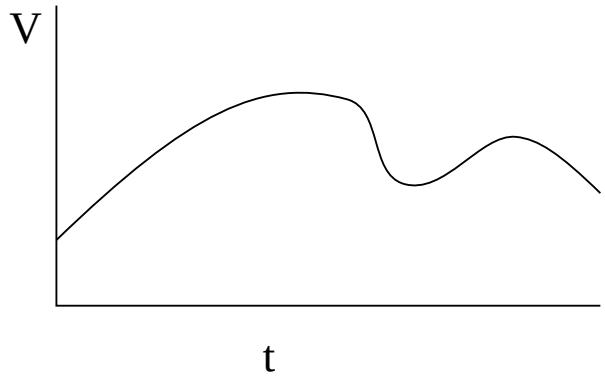
- Computers work with current/voltage that are related as $V = IR$, where V is voltage, I is current and R is resistance

Think About It

Assuming voltage (V) is **fixed**

- a low resistance (R) in the circuit implies high current(I), and
- a high resistance (R) in the circuit implies low current(I)

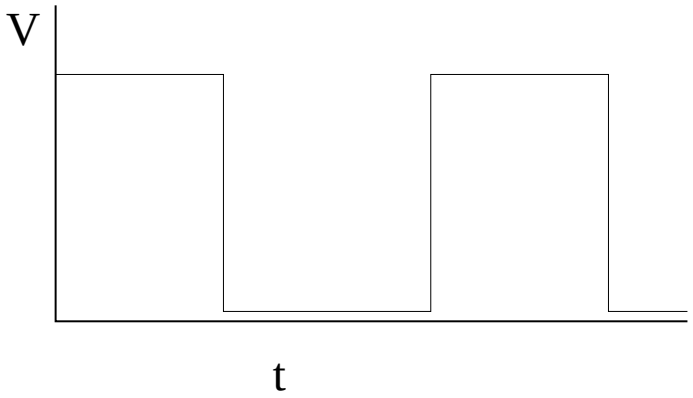
- These quantities are continuous, e.g. plot of voltage (V) vs time (t).



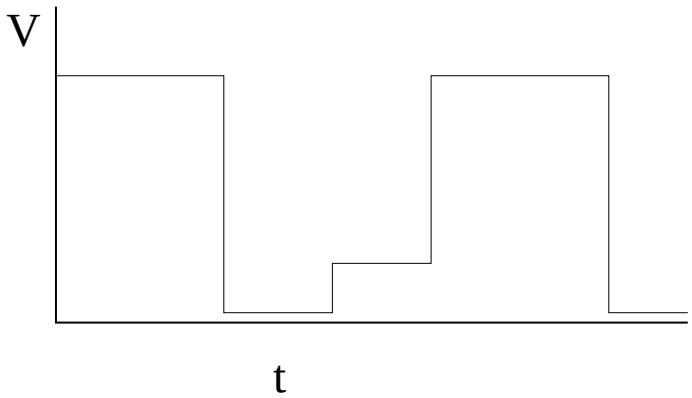
- We can "digitize" the analog signals.

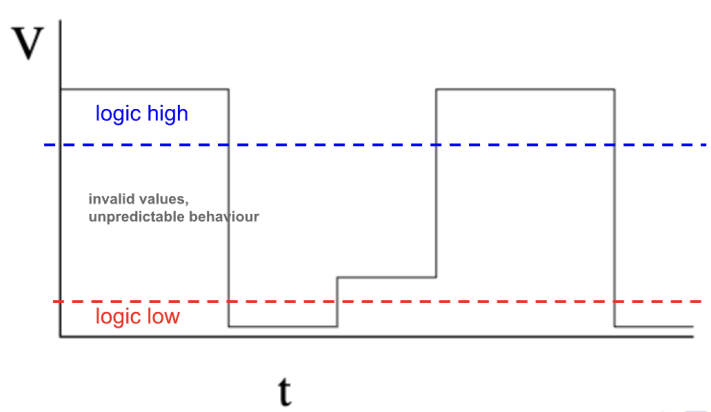
Digitizing

- Digital Signal



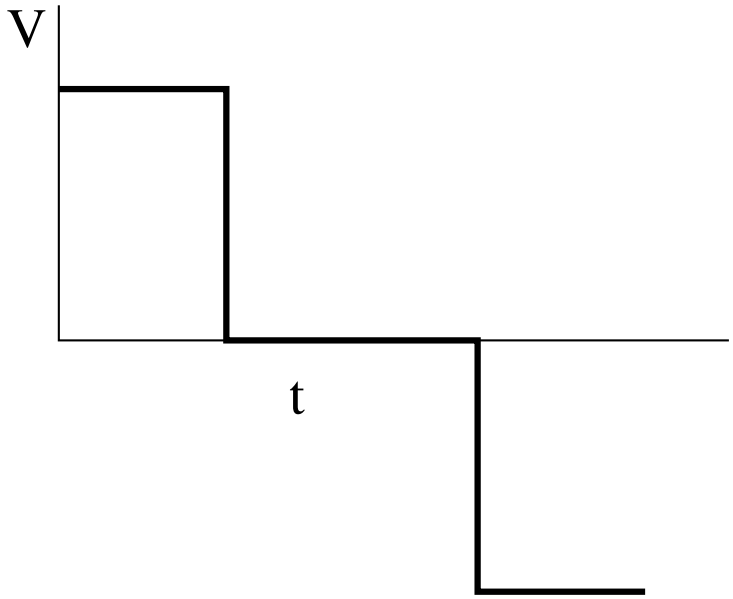
- Signal is either high (1) or low (0)
- Transformation could lead to intermediate values but these can be "designed out"





Why Binary?

- Could have more levels...

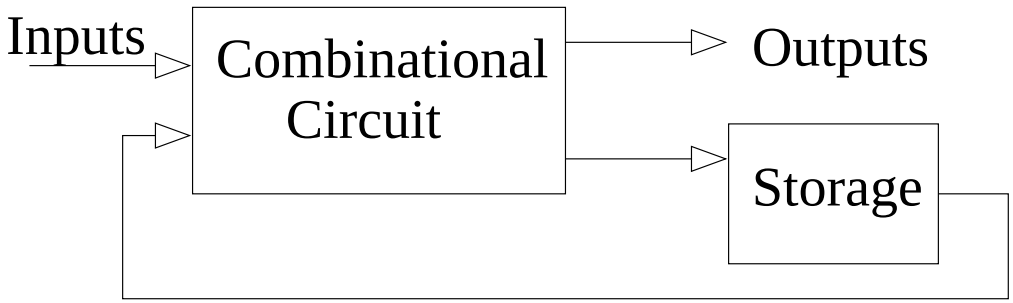


- Plus, Zero, Minus ($+1$, 0, -1 — ternary)
- Two levels are simpler and just as expressive
- Nearly all computers today use binary
- Nearly all computers have similar underlying structure

Logic Blocks

- Inputs and outputs are 1/0
  (High/low voltage, true/false)
- Combinational: without memory

- Sequential: with memory



Defining a Circuit as a Boolean Function

- Truth table: specifies outputs for each possible input combination

X	Y	Z	F	G
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

Remember It

- Inputs (X, Y, Z) and Outputs (F, G) are alphabetized
- For sub-scripted inputs or outputs, use numerical ordering based on index in scripts (A_2, A_1, A_0)
- Permutation of input values are in increasing order (000, 001, 010, 011...)

- Complete description: but can grow quickly & gets hard to understand

Compact Alternative: Boolean Expressions

- Define Boolean functions using Boolean expressions
- With Boolean inputs and operators
- Some *simple* Boolean operators and their truth tables:
  - OR ($+$) operator has result 1 *if* **either** input has value 1
  - AND ($\cdot$) operator has result 1 *iff* **both** inputs have value 1
    $A \cdot B$ often written AB
  - NOT ($\neg$) operator has result 1 *if* the input has value 0
    $\neg A$ usually written $\bar{A}$

OR		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

AND		
A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

NOT	
A	$\neg$A
0	1
1	0

Boolean Expressions Example 1

Think About It

To define the Boolean functions F and G using Boolean expressions is not always straightforward.

X	Y	Z	F	G
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

- It maybe easy to observe that $G = \bar{X} + \bar{Y} + \bar{Z} \equiv \overline{XYZ}$
- However, defining F is not straightforward.

Two-Level Representation for Boolean Expressions

- Any Boolean expression can be represented as a sum of products[1]
- level 1: AND logic followed by level 2: OR logic
- think about it as **OR** of **AND**s
- Each product in the sum corresponds to a single line in the truth table with output value 1
- **If** each product contains **all the input literals**, it is called a minterm
- The sum of products (SOP) is also the sum of the minterms

Try this

Define the Boolean function F using sum of products form.

X	Y	F
0	0	1
0	1	0
1	0	1
1	1	1

[1] An alternative is product of sums that can also be useful

From Truth Table To Boolean Expressions

- Realize that two-level representation is a systematic approach to define a Boolean function F
- Simply:
  - select the inputs in a logical AND operation for when output $F = 1$,
  - these are minterms that partially define F,
  - collect the minterms in a logical OR operation to completely define F.

A	B	C	F	$\bar{A}\bar{B}C$	$A\bar{B}C$	ABC	$\bar{A}\bar{B}C + A\bar{B}C + ABC$
0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	1
0	1	0	0	0	0	0	0
0	1	1	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	0	1
1	1	0	0	0	0	0	0
1	1	1	1	0	0	1	1

Don't Cares in Truth Tables

- A useful technique for compressing truth tables
- Represent value of a literal as X instead of 0 or 1
- When used in output: we don't care what the output is for that input combination
- When used in input: outputs are valid for all inputs created by replacing X by 0 or 1

Consider these two rows in the truth table below that give the same output value for F

A	B	C	F
0	0	0	0
0	0	1	0

Using don't cares for **input** C simplifies the truth table to:

A	B	C	F
0	0	X	0

Examples - Don't Cares in Truth Tables

Try this

Compress this truth table using don't cares.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Be Careful with Don't Cares

Think About It

Is the following truth table valid?

A	B	C	F
0	0	X	0
0	X	1	1
1	X	X	X

No. It is easy to write an incorrect specification.
Here are the problems with the above truth table:

- 001 appears in two rows with different outputs. A truth table must have exactly one output for each input.
- 010 is missing in the input. Each possible input must be specified exactly once.

Example: Boolean Expression for Truth Tables with Don't Cares

Try this

Write the Boolean expression for F

A	B	C	F
0	0	X	0
0	1	X	1
1	X	X	X

Compressed Truth Tables and Non-Minimal Terms

Remember It

To write the Boolean function F using compressed truth table, we can use minterms and, or non-minimal terms.

A	B	C	F	$\bar{A}\bar{B}C$	AC	$\bar{A}\bar{B}C + AC$
0	0	0	0	0	0	0
0	0	1	1	1	0	1
0	1	X	0	0	0	0
1	X	0	0	0	0	0
1	X	1	1	0	1	1

Using Overlapping Non-Minimal Terms

Think About It

To write the Boolean function F, we can use non-minimal terms that overlap. For example in the last row in the truth table.

A	B	C	F	AB	AC	$AB + AC$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	1	1	0	1	1
1	1	0	1	1	0	1
1	1	1	1	1	1	1

Can minterms overlap?

Boolean Expression Simplification

- Boolean expressions can often be simplified either manually or by computers.
- Take for example the following truth table

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

- Recall, $F = \bar{X}\bar{Y}Z + X\bar{Y}Z + XY\bar{Z} + XYZ$
- This is equivalent to $F = \bar{Y}Z + XY$

- Consider using don't cares and observe that:
- in last two rows, Z can be replaced with don't care, which implies, that F can be partially defined by XY.
- in row 2 and row 6, when $F = 1$, X can be replaced with a don't care and the non-minimal term for those two rows is $\bar{Y}Z$
- Therefore, $F = \bar{Y}Z + XY$

Laws of Boolean Algebra

- We can use algebraic manipulation based on laws of Boolean Algebra to simplify formulas.

Rule	Dual Rule	
$\overline{\bar{X}} = X$		
$X + 0 = X$	$X \cdot 1 = X$	(identity)
$X + 1 = 1$	$X \cdot 0 = 0$	(zero/one)
$X + X = X$	$XX = X$	(absorption)
$X + \bar{X} = 1$	$X\bar{X} = 0$	(inverse)
$X + Y = Y + X$	$XY = YX$	(commutative)
$X + (Y + Z) = (X + Y) + Z$	$X(YZ) = (XY)Z$	(associative)
$X(Y + Z) = XY + XZ$	$X + YZ = (X + Y)(X + Z)$	(distributive)
$\overline{X + Y} = \overline{X} \cdot \overline{Y}$	$\overline{XY} = \overline{X} + \overline{Y}$	(DeMorgan)

Boolean Expression Simplification Using Laws of Boolean Algebra

Recall, the truth table

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$
\begin{aligned}
F &= \bar{X}\bar{Y}Z + X\bar{Y}Z + XY\bar{Z} + XYZ \\
&\quad \text{use distributive law} \\
&= \bar{Y}Z(\bar{X} + X) + XY(\bar{Z} + Z) \\
&\quad \text{use inverse law} \\
&= \bar{Y}Z(1) + XY(1) \\
&\quad \text{use identity law} \\
&= \bar{Y}Z + XY
\end{aligned}
$$

- Difficult even for humans, tricky to automate
- Seems inherently hard to get "simplest" formula
- Is simplest formula the best for implementation?

Example: Simplifying Boolean Expressions

- Use Don't Care or Laws of Boolean Algebra to simplify Boolean expressions.
- Until the expression cannot be simplified any further.

Try this

Write a simplified Boolean function for F. Given that the truth table for F is as follows:

A	B	C	F	$\bar{A}\bar{B}C$	$A\bar{B}C$	ABC	$\bar{A}\bar{B}C + A\bar{B}C + ABC$
0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	1
0	1	0	0	0	0	0	0
0	1	1	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	0	1
1	1	0	0	0	0	0	0
1	1	1	1	0	0	1	1

Conclusion

Lecture Summary

- Defining digital functions
- Truth Tables
- Boolean expressions
- Sum of Product (SOP)
- Simplifying Boolean expressions using
  - Laws of Boolean Algebra
  - Don't Care
  - Nonminimal terms

Assigned Textbook Readings

- Sections A.1 and A.2 in Appendix A.

Next Steps

- **Ask** questions in the next tutorial or office hours.