CSE 2021
Computer Organization

Hugh Chesser,
CSEB 1012U

W5-M

Floating Point Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	`$f2 ← $f4+$f6`	Single Prec.
	FP subtract single	`sub.s $f2,$f4,$f6`	`$f2 ← $f4-$f6`	Single Prec.
	FP multiply single	`mul.s $f2,$f4,$f6`	`$f2 ← $f4×$f6`	Single Prec.
	FP divide single	`div.s $f2,$f4,$f6`	`$f2 ← $f4/$f6`	Single Prec.
	FP add double	`add.d $f2,$f4,$f6`	`$f2 ← $f4+$f6`	Double Prec.
	FP subtract double	`sub.d $f2,$f4,$f6`	`$f2 ← $f4-$f6`	Double Prec.
	FP multiply double	`mul.d $f2,$f4,$f6`	`$f2 ← $f4×$f6`	Double Prec.
	FP divide double	`div.d $f2,$f4,$f6`	`$f2 ← $f4/$f6`	Double Prec.
Data Transfer	load word FP Single	`lwc1 $f2,100($s2)`	`$f2 ← Mem[$s2+100]`	Single Prec.
	store word FP Single	`swc1 $f2,100($s2)`	`Mem[$s2+100] ← $f2`	Single Prec.

Conditional branch	FP compare single
	(eq, ne, lt, le, gt, ge)
		`c.lt.s $f2,$f4`	`if($f2<$f4)cond = 1,`
			`else cond = 0`
				Single Prec.

	FP compare double
	(eq, ne, lt, le, gt, ge)
		`c.lt.d $f2,$f4`	`if($f2<$f4)cond = 1,`
			`else cond = 0`
				Double Prec.

	Branch on FP true	`bc1t 25`	`if cond==1 go to`
			`PC+100+4`
				Single/
				Double Prec.

	Branch on FP false	`bc1f 25`	`if cond==0 go to`
			`PC+100+4`
				Single/
				Double Prec.

Example

```
# calculate area of a circle
         .data
Ans:     .asciiz      "The area of the circle is: "
Ans_add: .word        Ans                          # Pointer to String (Ans)
Pi:      .double      3.1415926535897924
Rad:     .double      12.345678901234567
Rad_add: .word        Rad                          # Pointer to float (Rad)
         .text
main:    lw $a0, Ans_add($0)                       # load address of Ans into $a0
         addi $v0, $0, 4                           # Sys Call 4 (Print String)
         syscall
#----------------                                  # load float (Pseudoinstruction)
         la $s0, Pi                                # load address of Pi into $s0
         ldc1 $f2, 0($s0)                          # $f2 = Pi
#----------------                                  # load float (MIPS Instruction)
         lw $s0, Rad_add($0)                       # load address of Rad into $s0
         ldc1 $f4, 0($s0)                          # $f4 = Rad
         mul.d $f12, $f4, $f4
         mul.d $f12, $f12, $f2
         addi $v0, $0, 3                           # Sys Call 3 (Print Double)
         syscall
exit:    jr $ra
```

Agenda for Today

1. Floating Point – Round off
2. Introduction to Hardware – Logic Design

Patterson: Section 3.5, Appendix C

Floating Point Round off

Floating Point arithmetic operations can lead to overflow (like integer arithmetic) and underflow

- Overflow – value is too large to be represented by the precision chosen (single or double)
- Underflow – value is too small to be represented by the precision chosen
- This situation leads to an exception – program/user is alerted (usually by an error message)
- What happens when the answer takes on a value that is between the floating point values that can be represented?

Ex – Floating Point Addition

Add: 9.999_{ten} x 10^1 and 1.610_{ten} x 10^{-1} (assume 3 digits of precision only)

$$9.99900 \times 10^1$$

$$\underline{0.01610 \times 10^1}$$

$$10.01510 \times 10^1 = \boxed{1.001}\boxed{5}\boxed{1} \times 10^1 = 1.002 \times 10^1$$

IEEE 754 specifies three extra digits for representation of FP calculations – "guard" and "round" – 2 bits used for multiplication operation

- > 50 – round up, <50 round down, =50?

Rounding modes: always round up, always round down, truncate, round to nearest even

Third bit – "sticky" – set when there are digits to the right of the round bit

Hardware – Logic Design

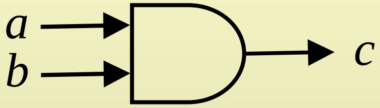
Appendix C goes through the basics of logic devices and how they implement the instructions we have been talking

Reference is made to the "Verilog" hardware description language (HDL)

- HDL – allows the "designer" (not programmer) to configure all of the programmable logic gates in a FPGA, ASIC or similar device
- HDL is "synthesized" (not compiled) to give a "netlist" (not machine code) which is downloaded to the device
- As the name suggests, HDL describes how the resulting logic circuits will manipulate "signals" (not variables)

Logical Operations: AND, OR, NOT, Multiplexer

1. AND Gate:



Symbol

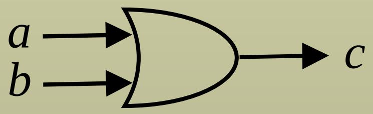
$(c = a \cdot b)$

Notation

a	b	$c = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table

2. OR Gate



Symbol

$(c = a + b)$

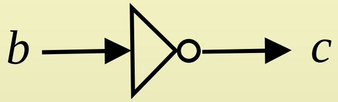
Notation

a	b	$c = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

Truth Table

Logical Operations: AND, OR, NOT, Multiplexer

3. NOT Gate (Inverter):



Symbol

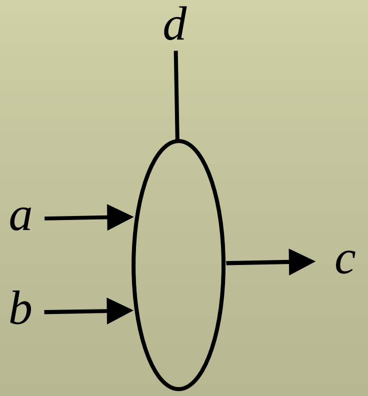
$(c = \overline{a})$

Notation

a	*c*
0	**1**
1	**0**

Truth Table

4. Multiplexer



Symbol

if $(d == 0),\ c = a;$

else $c = b;$

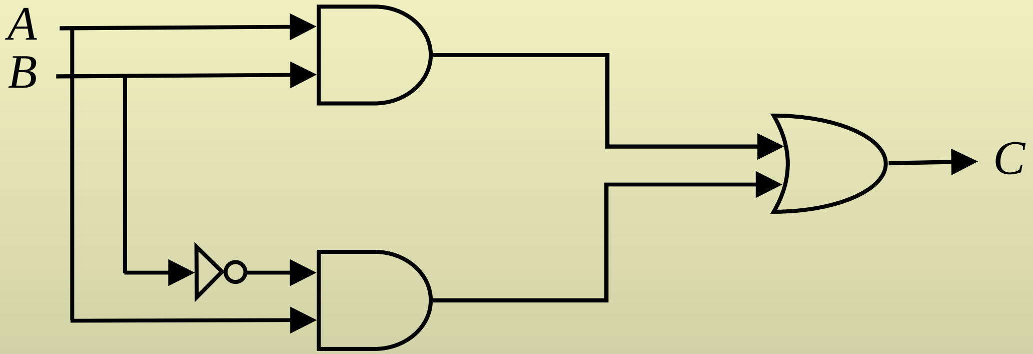
Notation

d	*c*
0	***a***
1	***b***

Truth Table

Boolean Algebra (1)

1. Logic Operations can be expressed in terms of logic equations



2. For the above figure, the output $C = AB + A\overline{B}$
3. To implement the above digital circuit, 2 AND, 1 NOT and 1 OR gates are required
4. Can we simplify the above circuit?

Boolean Algebra (1)

	Expressions
Identity Law	$A + 0 = A$
	$A \cdot 1 = A$
Zero and One Law	$A + 1 = 1$
	$A \cdot 0 = 0$
Inverse Law	$A + \bar{A} = 1$
	$\bar{A} \cdot 0 = 0$
Commutative law	$A + B = B + A$
	$A \cdot B = B \cdot A$
Associative Law	$A + (B + C) = (A + B) + C$
	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
Distributive Law	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
	$A + (B \cdot C) = (A + B) \cdot (A + C)$
DeMorgan Law	$\overline{(A + B)} = \overline{A} \cdot \overline{B}$
	$\overline{(A \cdot B\ )} = \overline{A} + \overline{B}$

Boolean Algebra (2)

Activity 1:

Simplify the expressions:

$$\begin{array}{ll}
(a) & \bar{A}B + ABC + AB\bar{C} \\
(b) & \bar{x}yz + xz \\
(c) & (\bar{x} + \bar{y})(\overline{x + y}) \\
(d) & xy + x(wz + w\bar{z}) \\
(e) & (B\bar{C} + \bar{A}D)(A\bar{B} + C\bar{D})
\end{array}$$

Activity 2:

Implement simplified expressions for (a) – (e) using OR, AND, and NOT gates

Combinational Logic: Design of a 1-bit adder (1)

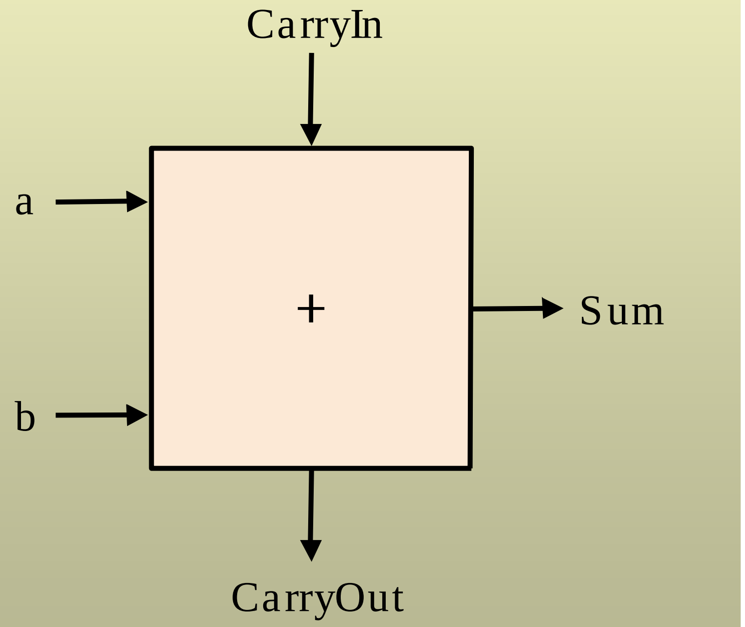
Example: Design an 1-bit adder with Carry-in

Step 1: Construct the truth table for an 1-bit adder

3 binary inputs imply ($2^3 = 8$) entries in the truth table

INPUTS			OUTPUTS	
a	**b**	**c (CarryIn)**	**CarryOut**	**Sum**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
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

Truth Table for 1-bit adder



Schematic of a 1-bit adder

Combinational Logic: Design of a 1-bit adder (2)

Step 2: Derive the Boolean expression for each output from the truth table

INPUTS			OUTPUTS	
a	b	c (CarryIn)	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
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

$$\mathrm{Sum} = \overline{a}\,\overline{b}\,c + \overline{a}\,b\,\overline{c} + a\,\overline{b}\,\overline{c} + abc$$

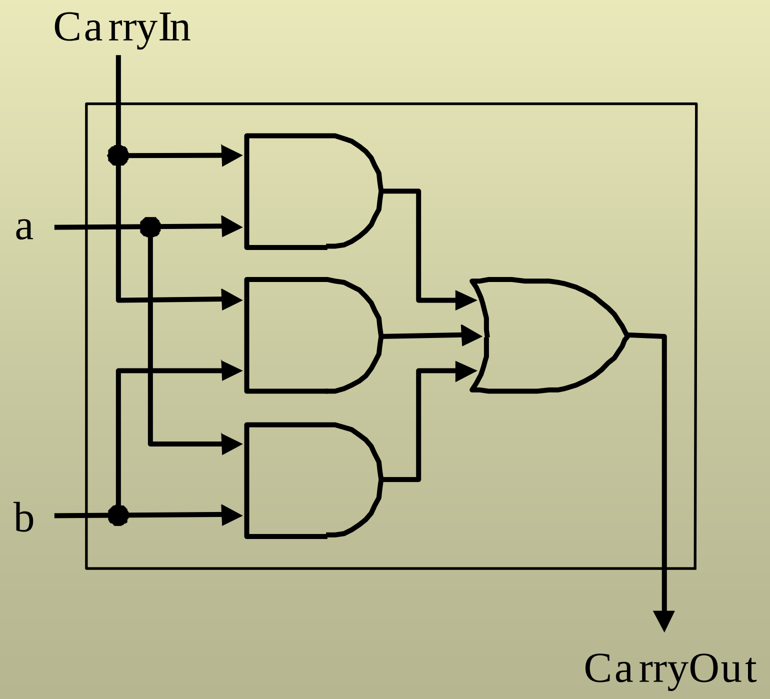
$$\mathrm{Carryout} = \overline{a}\,bc + a\,\overline{b}\,c + ab\overline{c} + abc$$

Combinational Logic: Design of a 1-bit adder (3)

Step 3: Simplify the Boolean expression

$$\text{Carryout} = \bar{a}bc + a\bar{b}c + ab\bar{c} + abc = bc + ac + ab$$

Step 4: Implement the simplified Boolean expression using OR, AND, and NOT gates



Activity: Implement the hardware for the Sum output of the 1-bit adder