Lecture 10: ALU

Topics

- Minimising Boolean expressions
  - Using Karnaugh maps
- ALU (Arithmetic Logic Unit)
- ALU design and implementation
  - 1-bit ALU
  - 32-bit ALU

SONGS ABOUT COMPUTER SCIENCE

…COMPUTER SCIENCE MAJOR?
Written by Emmanuel Schanzer
To the tune of: Hotel California
http://www.cs.utexas.edu/users/walter/cs-songbook/digital_logic.html

… …
My mind is completely twisted
My brain's completely snapped
By these logic gates and Turing machines
And those **Karnaugh maps**
Registers dance in memory
Clobbering the temps
Some values you remember
Some values you forget
So I called up the professor
Can I have more time?
He said
I haven't given an extension here since 1969
… …

Building blocks revisited

- We will build ALU using four hardware building blocks:

1. **AND** gate (c = a • **b)**

a	b	c = a . b
0	0	0
0	1	0
1	0	0
1	1	1

2. **OR** gate (c = a + b)

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

3. **Inverter** (c = a)

a	c = $\bar{a}$
0	1
1	0

4. **Multiplexor** (Mux) (if d = = 0, c = a; else c = b)

d	c
0	a
1	b

Minimising Boolean expressions

- Before we start building ALU, consider how to minimise logic expressions in easy way, and implement circuits with as few logic gates as possible.
- For example, soon we will see that Carry Out formula expressed as **a sum of products** is (to be explained later):
  CarryOut = (A'*B*CarryIn) + (A*B'*CarryIn) + (A*B*CarryIn') + (A*B*CarryIn)
  …happens to be equivalent of:
  CarryOut = (B*CarryIn) + (A*CarryIn) + (A*B)
- But: the above simplification is not immediately obvious.
- Logic minimising tool which we will use is known as:
  Karnaugh maps.

Karnaugh maps

- Product items: Miniterms

3-Input Function (inputs a, b, c; output y)

On [b] / Off [b']

a\bc	b	b	b'	b'
a	abc'	abc	ab'c	ab'c'
a'	a'bc'	a'bc	a'b'c	a'b'c'

----b---- ----c----

2^3 = 8 cases

- Sum of Products $y=y_0\bullet abc'+y_1\bullet abc+y_2\bullet ab'c+y_3\bullet ab'c'+$ …
- Use **Truth Table** to determine $\mathbf{y_0, y_1, \ldots}$
- Use **Karnaugh maps** (or **K-maps**) to simplify the expression

Karnaugh maps: [I] State Sets

- State Sets for 2, 3 and 4-variable functions [**a' stands for NOT a**]

a → a'

a\b	b	b'
a	ab	ab'
a'	a'b	a'b'

2^2 = 4 cases

a\bc	b	b	b'	b'
a	abc'	abc	ab'c	ab'c'
a'	a'bc'	a'bc	a'b'c	a'b'c'

----b---- ----c----

2^3 = 8 cases

ac\bd	b	b	b'	b'
a	abc'd'	abc'd	ab'c'd	ab'c'd'
a	abcd'	abcd	ab'cd	ab'cd'
	a'bcd'	a'bcd	a'b'cd	a'b'cd'
	a'bc'd'	a'bc'd	a'b'c'd	a'b'c'd'

----b---- ----a---- ----c---- ----d----

2^4 = 16 cases

Karnaugh maps: [II] Truth Table

- Truth Tables
  - determined by the internal functions

AND (inputs a, b; output y) ⇔

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

a\b [Truth table for AND]

	b [on]	b' [off]
a	1	0
a'	0	0

2-Input Function

a\b [State Set]

	b	b'
a	ab	ab'
a'	a'b	a'b'

$$y_{AND}=y_0\bullet ab+y_1\bullet ab'+y_2\bullet a'b+y_3\bullet a'b'$$
$$=1\bullet ab+0\bullet ab'+0\bullet a'b+0\bullet a'b'$$
$$=ab$$

2^2 = 4 cases $y=y_0\bullet ab+y_1\bullet ab'+y_2\bullet a'b+y_3\bullet a'b'$

Karnaugh maps: Simple mapping examples

- 2-input functions

2-Input Function

a\b [State Set]

	b	b'
a	ab	ab'
a'	a'b	a'b'

2^2 = 4 cases

$y=y_0\bullet ab+y_1\bullet ab'+y_2\bullet a'b+y_3\bullet a'b'$

	b	b'
a	0	0
a'	0	0

What logic function is that? out=0

	b	b'
a	1	1
a'	1	1

What logic function is that? out=1

	b	b'
a	1	1
a'	0	0

What logic function is that? out=ab+ab' out=a(b+b') out=a

Karnaugh maps: Grouping for simplification

- Rules of grouping:
  - of 1s
  - side by side
- Rules of simplification
  "A change of one variable when crossing a horizontal or vertical boundaries of cells"
  - Walk through a group
  - Invariables survive; changed ones eliminated
  - **Sum of net results of all groups (clusters)**

	b	b'
a	1	1
a'	0	0

OK out=a

	b	b'
a	1	1
a'	0	1

OK out=a+b'

	b	b'
a	1	0
a'	0	1

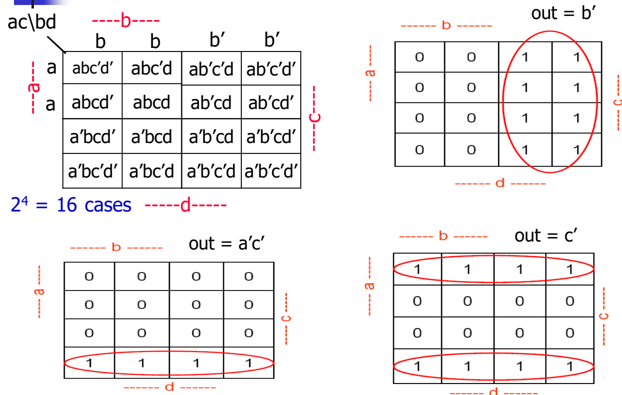
OK out=ab+a'b'

	b	b'
a	0	0
a'	0	1

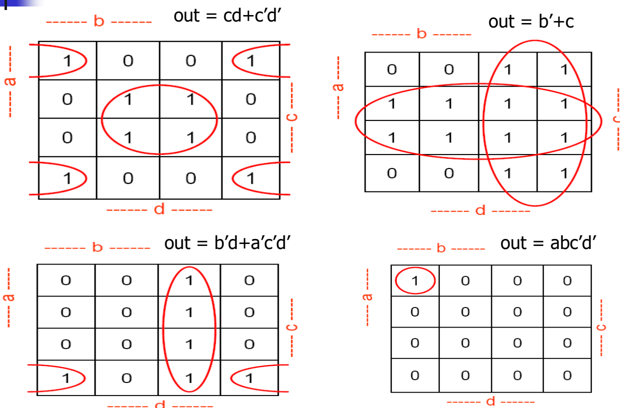
	b	b'
a	0	1
a'	1	0

WRONG!

Karnaugh maps: Grouping for simplification



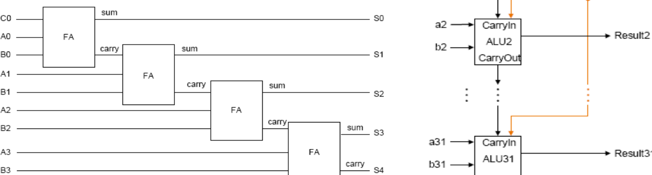
Karnaugh maps: Grouping for simplification



How to build ALU [Arithmetic Logical Unit]

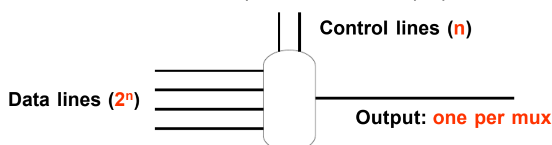
- 1-bit building blocks ready to implement, but MIPS word is 32 bits wide.
- Solution:
  - Build 32 separate 1-bit ALUs
  - Build separate hardware blocks for each task
  - Perform all operations in parallel
  - Use a mux to chose operations

A cascaded view of 4-bit 'Full Adder'



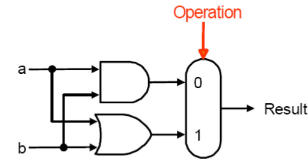
More about muxes

- Have **data** bits and **control** bits (data lines and control lines)
- Control bits select which data bit will pass through: all others are blocked
- In general:
  - 1 control bit selects between 2 data bits,
  - 2 control bits select between 4 data bits,
  - . . .
  - n control bits select between 2^n data bits
- We can build a mux of any size to serve our purpose



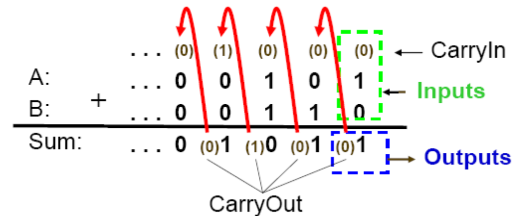
Build Logical Operations

- ALU = { Logical Functions: AND, OR ; Arithmetic Operations { Adder ; Subtraction } … …
- FIRST: Logical Functions
  - the easiest to implement, they map directly into the hardware
  - 1-bit logical block for AND and OR:
    - Mux control line Operation=0 selects a AND b
    - Mux control line Operation=1 selects a OR b



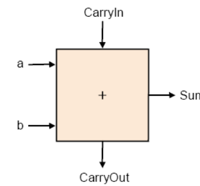
Build 1-bit Adder: Theory

- Each bit of addition has
  - Three input bits: A_i B_i $CarryIn_i$
  - Two output bits: Sum_i $CarryOut_i$
    ($CarryIn_{i+1}$ = $CarryOut_i$)



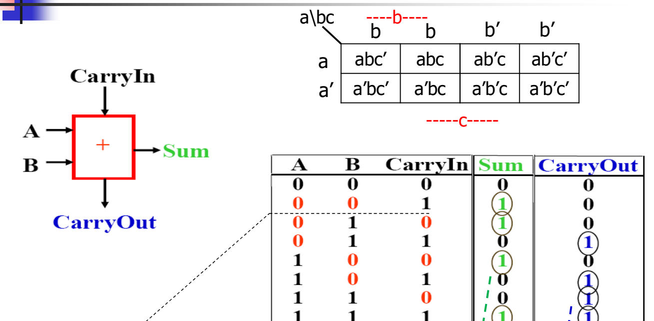
Build 1-bit Adder: implementation

- Full adder, also called a (3,2) adder: 3 inputs and 2 outputs
- Half adder, also called (2,2) adder has only 2 inputs, a and b.



- STEPS to implement the adder:
  1. Construct the circuit for Sum
  2. Construct the circuit for CarryOut
  3. Connect (1) and (2) together

1-bit Full Adder: truth table and formula



a\bc	b	b	b′	b′
a	abc′	abc	ab′c	ab′c′
a′	a′bc′	a′bc	a′b′c	a′b′c′

A	B	CarryIn	Sum	CarryOut
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- **Sum** =
  (A′*B′*CarryIn) + (A′*B*CarryIn′) + (A*B′*CarryIn′) + (A*B*CarryIn)
- **CarryOut** =
  (A′*B*CarryIn) + (A*B′*CarryIn) + (A*B*CarryIn′) + (A*B*CarryIn)

1-bit Full Adder: the Sum formula

- Can we simplify/minimise logic formulas for CarryOut and Sum for building the circuit using logic gates?
- Karnaugh table of the Sum formula: … *grouping 1s*

a\bc	b	b	b′	b′
a	0	1	0	1
a′	1	0	1	0

(-----b----- over the first two columns; -----c----- under the middle two columns)

No simplification possible

Sum=(A′*B′*CarryIn) + (A′*B*CarryIn′) + (A*B′*CarryIn′) + (A*B*CarryIn)

a\bc	b	b	b′	b′
a	abc′	**abc**	ab′c	**ab′c′**
a′	**a′bc′**	a′bc	**a′b′c**	a′b′c′

1-bit Full Adder: the CarryOut formula

- Karnaugh table of the CarryOut formula: … *grouping 1s*

a\bc	b	b	b′	b′
a	1	1	1	0
a′	0	1	0	0

(-----b----- over the first two columns; -----c----- under the middle two columns)

(B*CarryIn) + (A*CarryIn) + (A*B)

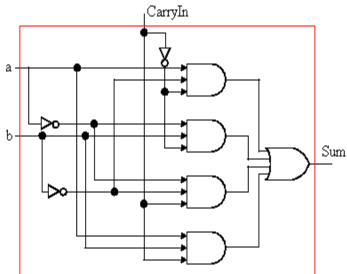
CarryOut
=(A′*B*CarryIn) + (A*B′*CarryIn) + (A*B*CarryIn′) + (A*B*CarryIn)
=(B*CarryIn) + (A*CarryIn) + (A*B)

	b	b	b′	b′
a	**abc′**	**abc**	**ab′c**	ab′c′
a′	a′bc′	**a′bc**	a′b′c	a′b′c′

1-bit Full Adder: formula -> circuit construction

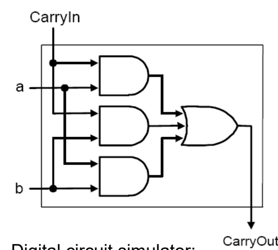
- For Sum

Sum=(A′*B′*CarryIn) + (A′*B*CarryIn′)
+ (A*B′*CarryIn′) + (A*B*CarryIn)



- For CarryOut

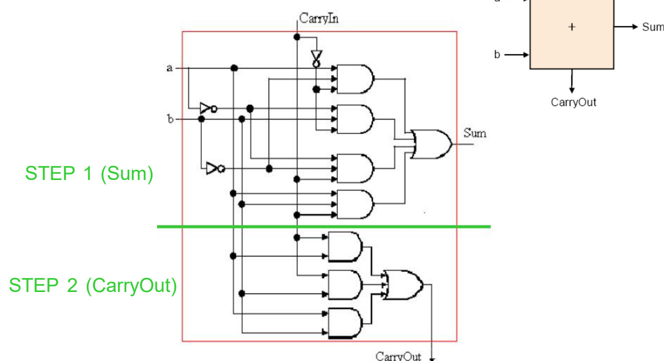
CarryOut=(B*CarryIn) +
(A*CarryIn) +
(A*B)



Digital circuit simulator: https://circuitverse.org/

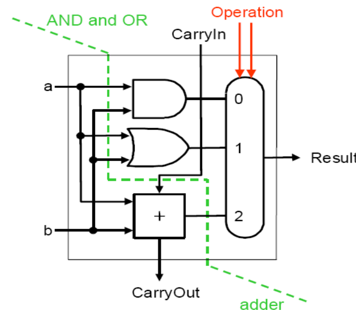
1-bit Full Adder: integrated circuit construction

- Connect Sum and CarryOut together



ALU: Adder and Logical operations

- ALU = { Logical Functions: AND, OR ; Arithmetic Operations { Adder, Subtraction } … …

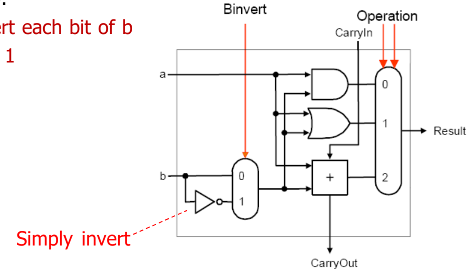


A 1-bit ALU that performs AND, OR and Addition.

Mux "Operation" signal selects which operation is performed.

Subtraction 1/3

- Subtraction: adding the negative version of an operand.
- Recall two's complement numbers: to create a negative number we need to:
  1. invert each bit of b
  2. add 1



Simply invert

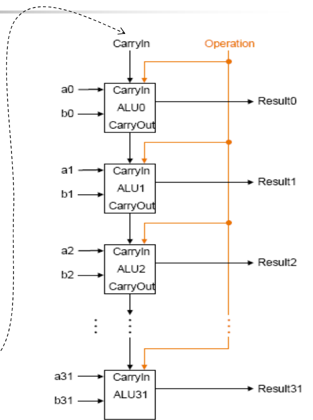
How about '`add 1`'?

Subtraction 2/3

- Notice that, the least significant bit still has CarryIn signal, which is never used for addition
- How CarryIn for ALU0 differs from other CarryIns?

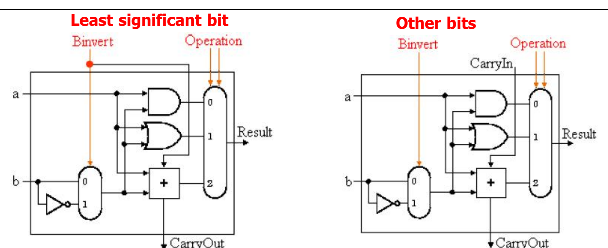
- Initial CarryIn vs. Intermediate CarryIns.
- Initial CarryIn can be set at will

- If we set the Initial CarryIn bit to 1 instead of 0, we get: a+b+1

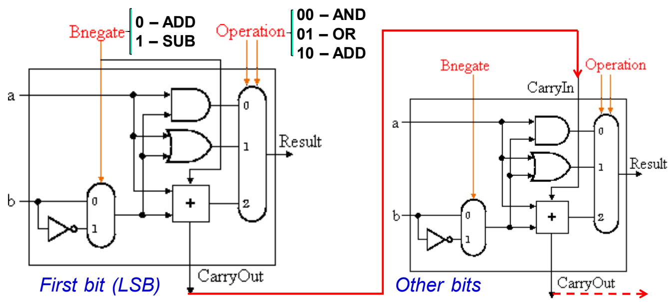


Subtraction 3/3

- What happens if we now use Binvert for invert b and set the Initial CarryIn (at the least significant bit) to 1?
  - The adder calculates: $a+\overline{b}+1 = a+(\overline{b}+1) = a+(-b)=a-b$
- This simplicity of hardware implementation of a two's complement adder is good illustration why two's complement representation is commonly used for integer computer arithmetics!

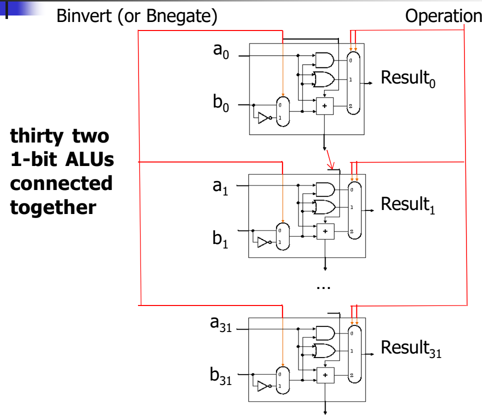


ALU structure so far



- Operations: AND OR ADD SUB
- Control lines: 000 001 010 110

32-bit ALU so far

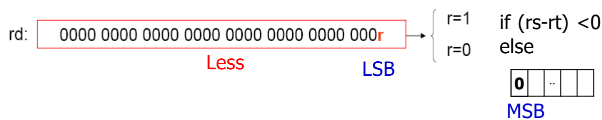


thirty two 1-bit ALUs connected together

Set on less than (slt) support

- slt rd, rs, rt

rd := { 1 if (rs < rt)
0 else }

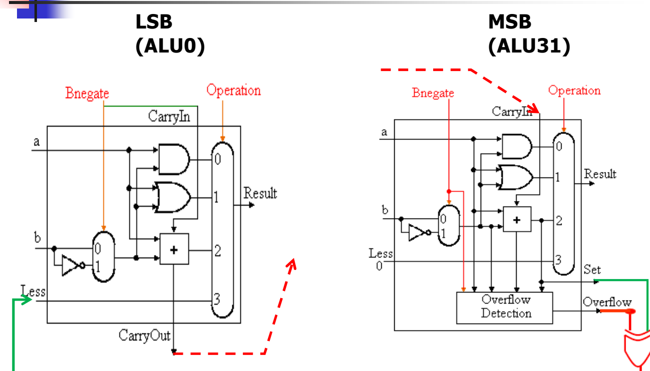


r=1 if (rs-rt) <0
r=0 else

MSB
[determines the sign]

- Implement the idea in ALU
  - modify 1-bit ALU for the most significant bit (ALU31 for bit 31):
    - a new output line (**Set** – 1 bit**)** used only for slt
    - (by the way we added overflow detection logic, also associated with this bit)
  - new input line (**Less** – 32 bits) goes directly to mux
  - New control line (111) for slt

ALU with slt support



Branch support

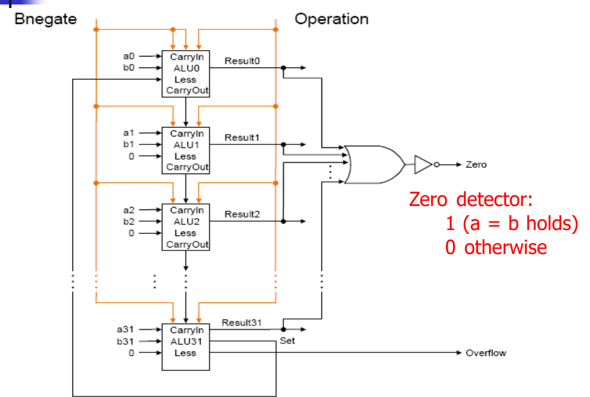
- Conditional branch instructions switch either if two registers are equal, or if they are not equal:
  - beq register1, register2, label (example: beq $s4,$s2, LABEL1)
  - bne register1, register2, label (example: bne $s1,$s3, L7)
- How to test that contents of two register is equal?
  - a = b means (a – b) = 0
- How to implement the above:
  - Subtract b from a
  - Add hardware to test if the result is zero
    - OR all the outputs together, and invert the output:

$$Zero = \overline{(Result1 + Result2 + .. + Result31)}$$

variable { 1 (true) if (a – b) = 0 holds
0 (false) else }

- Next slide show additional hardware for branch support :

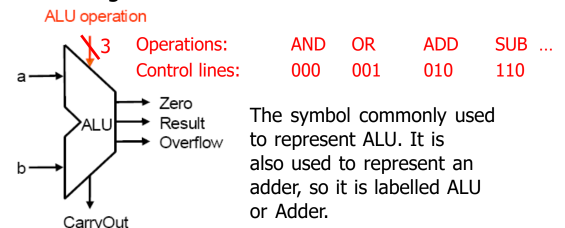
32-bit ALU with branch support



Zero detector:
1 (a = b holds)
0 otherwise

Shift instructions

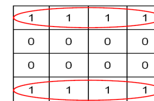
- What is left:
  - logical and arithmetic shifts sll, srl, sra
    - Need a new data line for a shifter (L and R)?
    - however shifters are much more easily implemented outside the ALU.
- 1-bit ALU: integrated block notation



The symbol commonly used to represent ALU. It is also used to represent an adder, so it is labelled ALU or Adder.

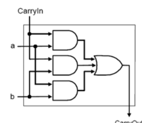
Revision and quiz

- Given the following Karnaugh map. The output can be expressed by out = c'.
  1) True 2) False



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

- In 1-bit full Adder, the circuit construction for 'CarryOut' is correct:
  1) True 2) False



- How ALU is used to support the branch instruction?
  beq register1, register2, label

Recommended readings

General Data	UnitOutline | LearningGuide | Teaching Schedule | Aligning Assessments |
Extra Materials	ascii_chart.pdf | bias_representation.pdf | HP_AppA.pdf | instruction decoding.pdf | masking help.pdf | PCSpim.pdf | PCSpim Portable Version | Library materials

PH6: Appendix B: The Basics of Logic Design
PH5: Appendix B: The Basics of Logic Design
PH4: Appendix C: The Basics of Logic Design

Text readings are listed in Teaching Schedule and Learning Guide

PH6 (PH5 & PH4 also suitable): check whether eBook available on library site

PH6: companion materials (e.g. online sections for further readings)

https://www.elsevier.com/books-and-journals/book-companion/9780128201091

PH5: companion materials (e.g. online sections for further readings)
http://booksite.elsevier.com/9780124077263/?ISBN=9780124077263