University of Nottingham
UK | CHINA | MALAYSIA

Digital Blocks and ALU Design

Dr. Heng Yu

AY2023-24, Spring Semester
COMP1047: Systems and Architecture
Week 4

Outline

Today's outline

- Basic digital building blocks
  - Gates, Arithmetic circuits, multiplexers, registers
- Constructing an arithmetic logic unit
  - Add, sub, and, or, nor
  - Set-on-less-than

Learning Objectives

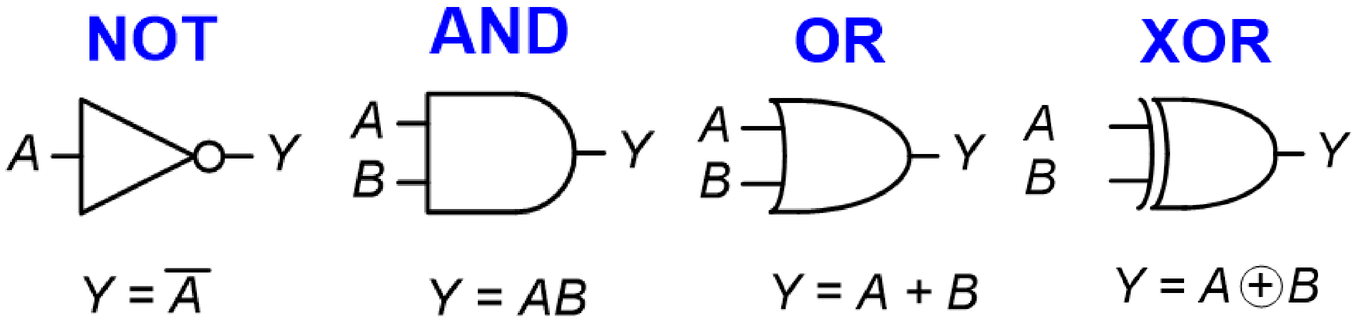
- Recap on basic digital circuit components
- Understand the function and control of a MIPS ALU
- Design a MIPS ALU with given arithmetic operations

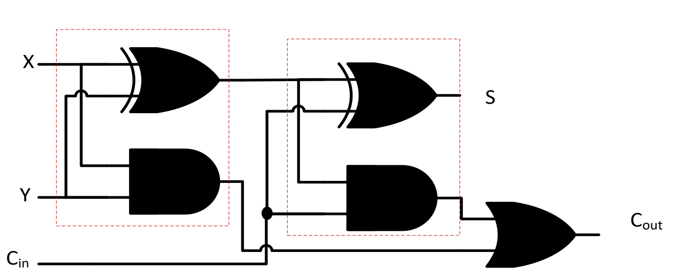
Introduction

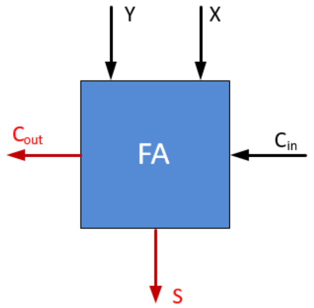
- Digital building blocks include:
  - Gates, arithmetic circuits, multiplexers, decoders, registers, counters, memory arrays, logic arrays, etc.

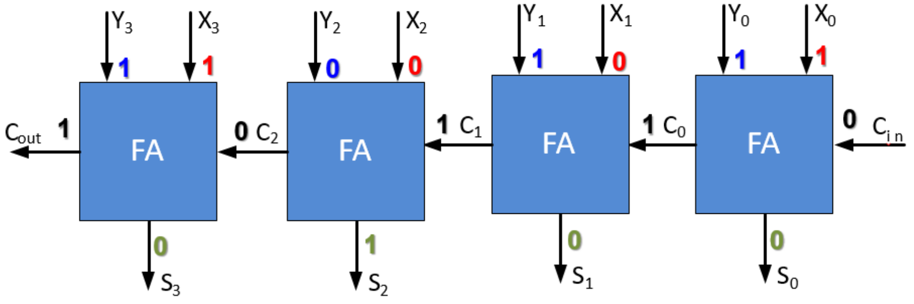
- Building blocks are important in their own right and they demonstrate hierarchy, modularity, and regularity:
  - They are built from a hierarchy of simpler components.
  - They have well-defined interfaces and functions.
  - Their regular structure is easily extended to different sizes.

- We'll use many of these building blocks to build a microprocessor in week 5.







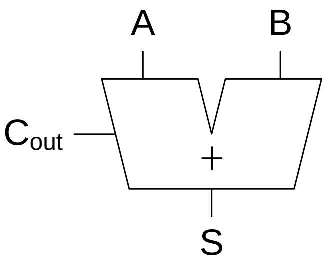


University of Nottingham
UK | CHINA | MALAYSIA

Digital Building Blocks

1-bit adders

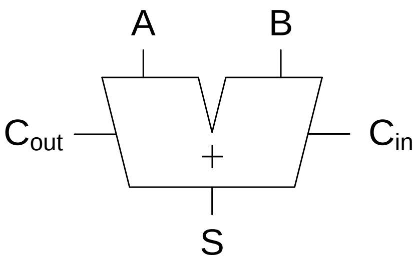
Half Adder



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$
$$C_{out} = AB$$

Full Adder



C_{in}	A	B	C_{out}	S
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

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Multi-bit adders

- Also called carry propagate adder (CPA). Several types of design available:
  - Ripple carry adder (slow)
  - Carry-Lookahead adder
  - Prefix adder
- Faster adders require more hardware.

Ripple carry adder

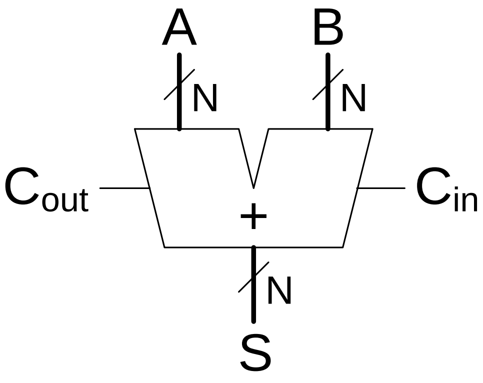
- Chain 1-bit adders together
- Disadvantage: slow
- The delay of an *N*-bit ripple-carry adder is:

$$t_{\text{ripple}} = N t_{FA}$$

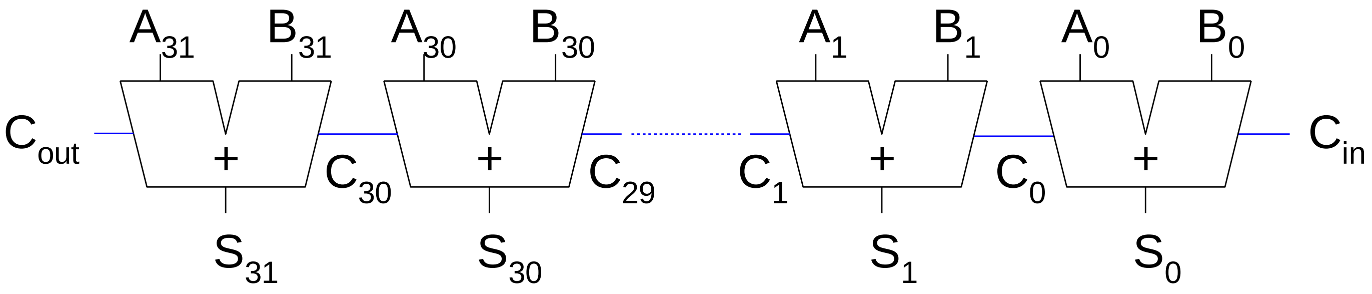
where t_{FA} is the delay of a full adder

Think about it:

Illustrate the process that a RCA works.



General symbol of a multi-bit adder.

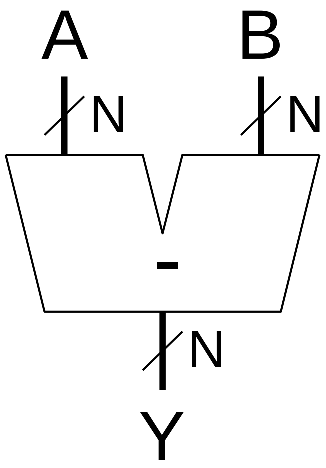


A 32-bit ripple carry adder.

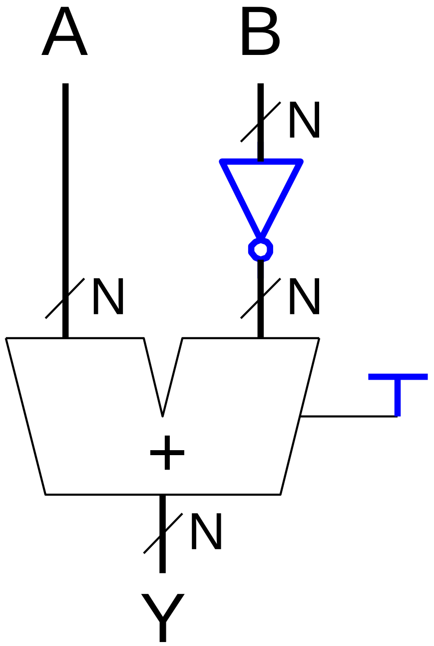
Subtractor

- – B = B’ + 1 (2’s complement representation)
- A – B = A + B’ + 1 (use CPA where Cin = 1)
- Bit-wise inverse of B is B’

Symbol

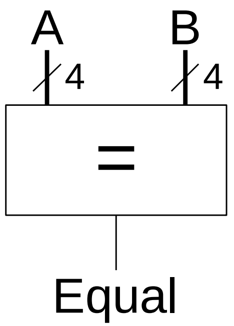


Implementation

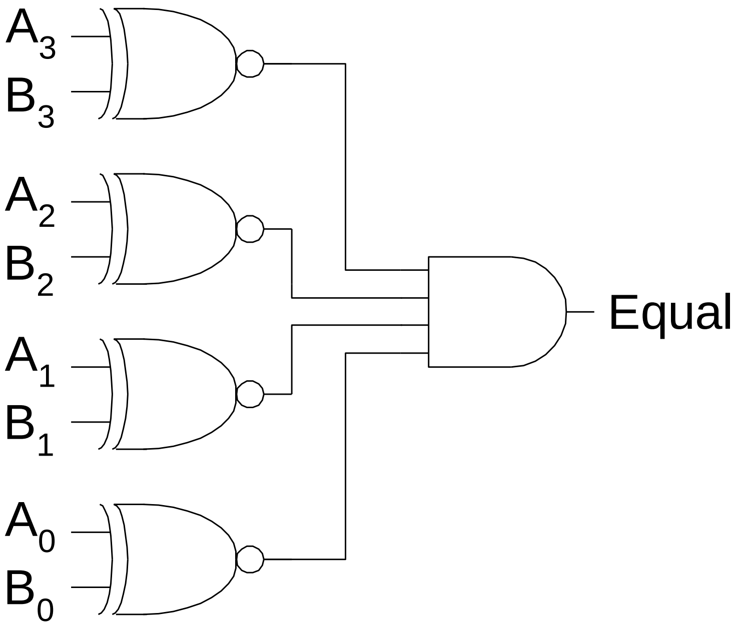


Comparator: Equality

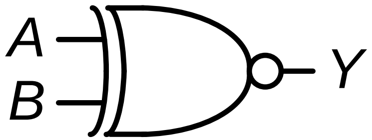
Symbol



Implementation



XNOR

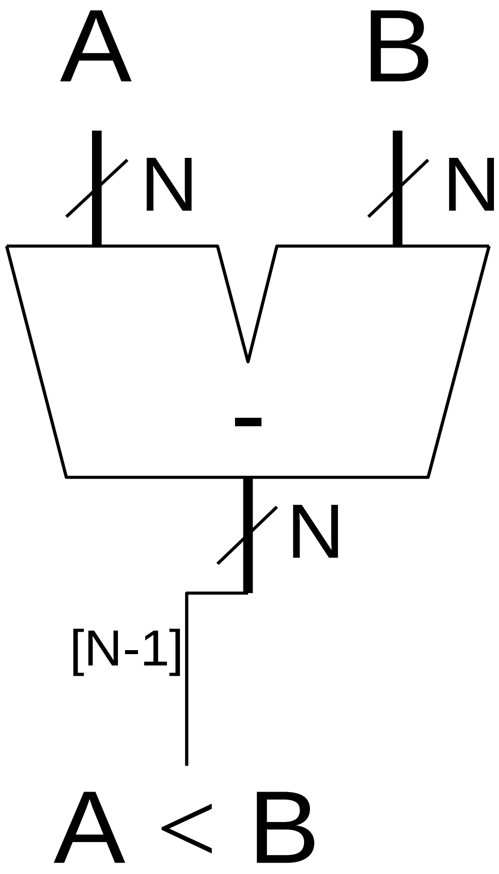


$$Y = \overline{A \oplus B}$$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

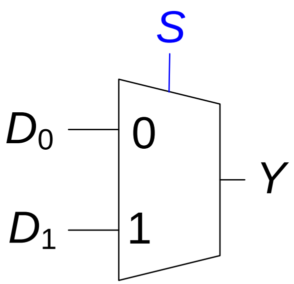
Comparator: Less than

- Calculate A – B. If A < B, then result is negative => 2's complement numbers, (N-1)th bit is 1.



Multiplexer

- Selects between one of N inputs to connect to the output.
- $\log_2 N$-bit input selection



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

S	Y
0	D_0
1	D_1

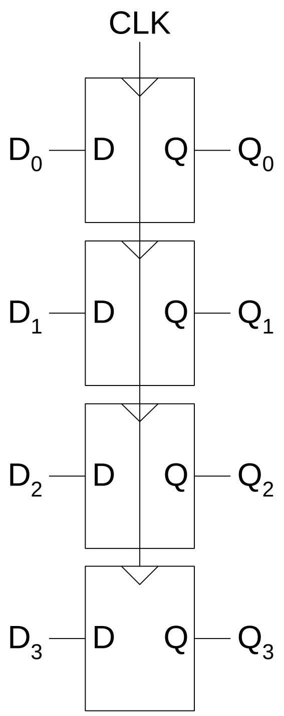
A 2-to-1 multiplexer

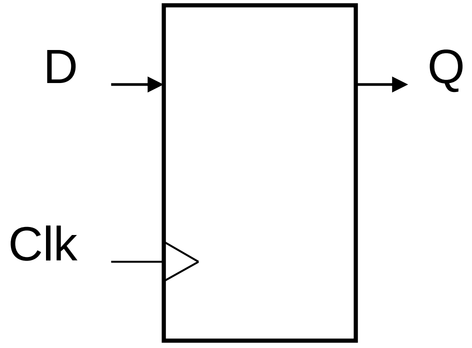
Think about it:

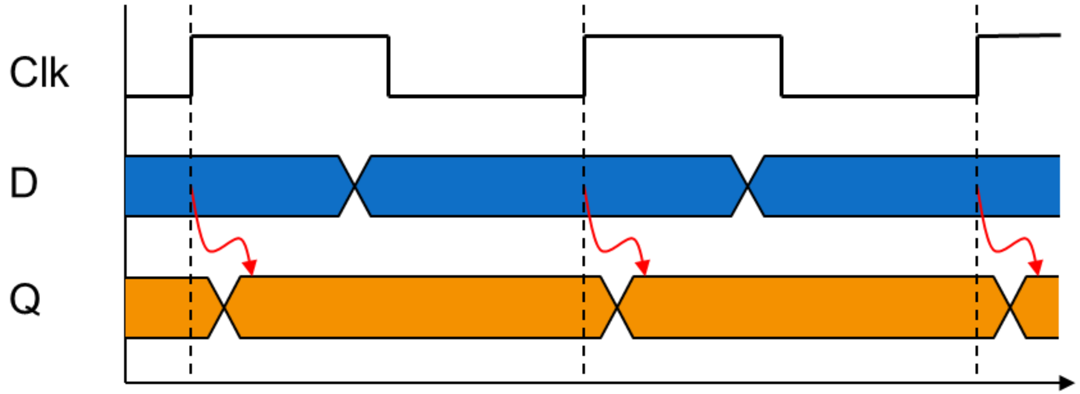
for 4-input mux, how many S bits are required? How about 8, 16, 32, …

Register

- Register: stores data in a circuit
- Uses a clock signal to determine when to update the stored value
- Edge-triggered: update when Clk changes from 0 to 1 (rising edge)
- Synchronous circuit







Today's outline

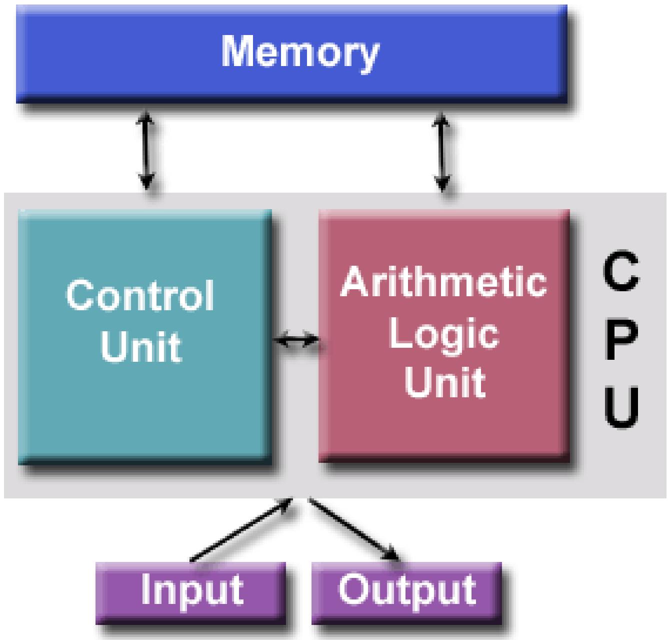
- Basic digital building blocks
  - Gates, Arithmetic circuits, multiplexers, registers
- Constructing an arithmetic logic unit
  - Add, sub, and, or, nor
  - Set-on-less-than

ALU Design

Define our ISA first!

Let's use a *subset* of the MIPS instructions.

- R-Type:
  - `add rd, rs, rt`
  - `sub rd, rs, rt`
  - `and rd, rs, rt`
  - `or rd, rs, rt`
  - `slt rd, rs, rt`
- Load/Store:
  - `lw rt,rs,imm16`
  - `sw rt,rs,imm16`
- Imm operand:
  - `addi rt,rs,imm16`
- Branch:
  - `beq rs,rt,imm16`



31 – 26	21	16	11	6	0
op	**rs**	**rt**	**rd**	**shamt**	**funct**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

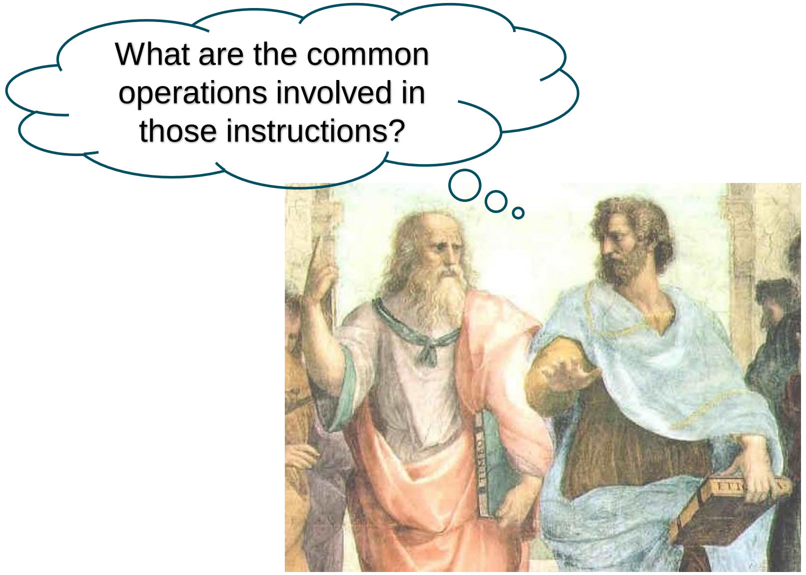
31 – 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

ALU Design

Define our ISA first!

Let's use a *subset* of the MIPS instructions.

- R-Type:
  - `add rd, rs, rt`
  - `sub rd, rs, rt`
  - `and rd, rs, rt`
  - `or rd, rs, rt`
  - `slt rd, rs, rt`
- Load/Store:
  - `lw rt,rs,imm16`
  - `sw rt,rs,imm16`
- Imm operand:
  - `addi rt,rs,imm16`
- Branch:
  - `beq rs,rt,imm16`



Requirements: must support the following arithmetic and logic operations

add, sub: two's complement adder/subtractor

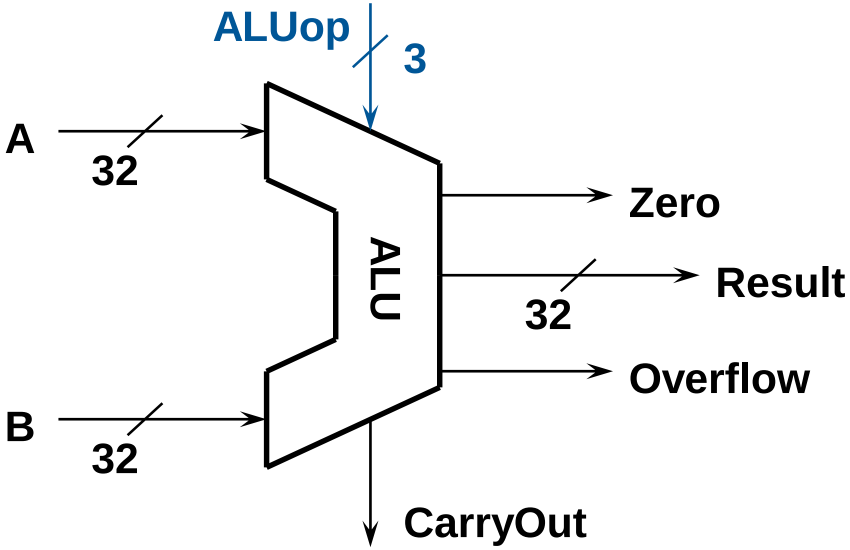
and, or : logical AND, logical OR

slt (set on less than): two's complement adder with inverter, check sign bit of result

ALU Design

Functional specification

Remember the operations needed:
add, sub, and, or, slt

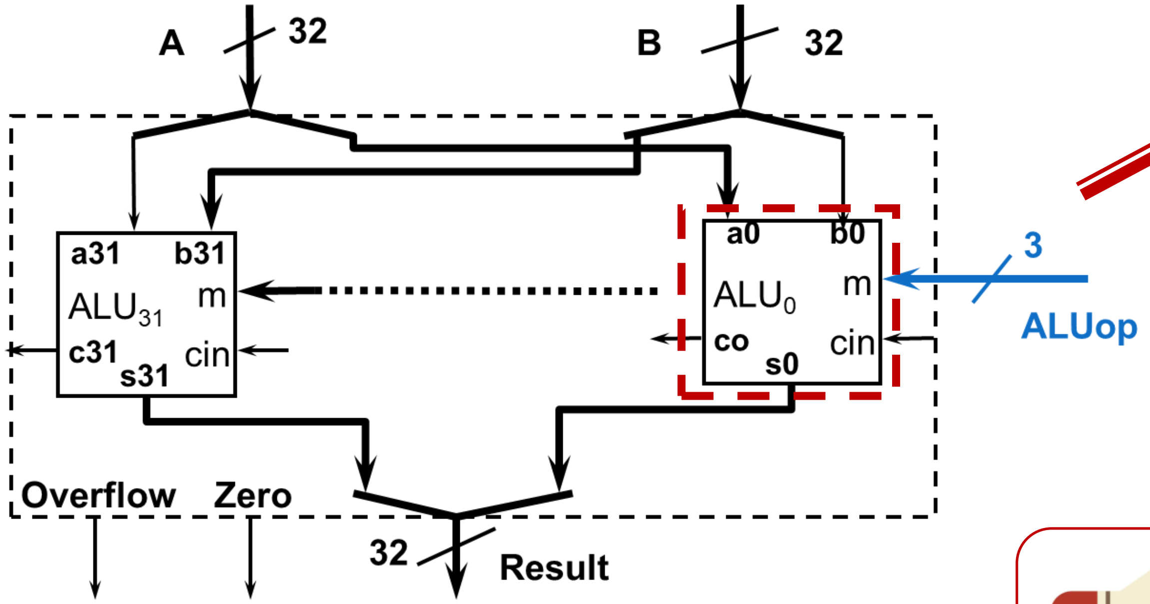


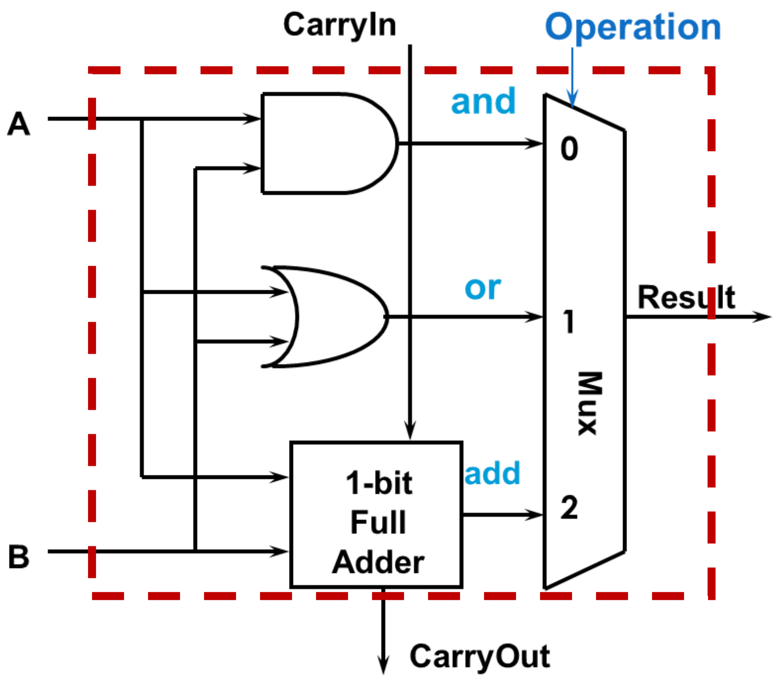
ALU Control (ALUop)	Operation
000	and
001	or
010	add
110	subtract
111	set-less-than

Why ALUop needs 3 bits?

A bit-slice ALU

- Design trick 1: divide and conquer
  - Break the problem (32-bit ALU) into simpler problems (1-bit ALU), solve them and glue together the solution.

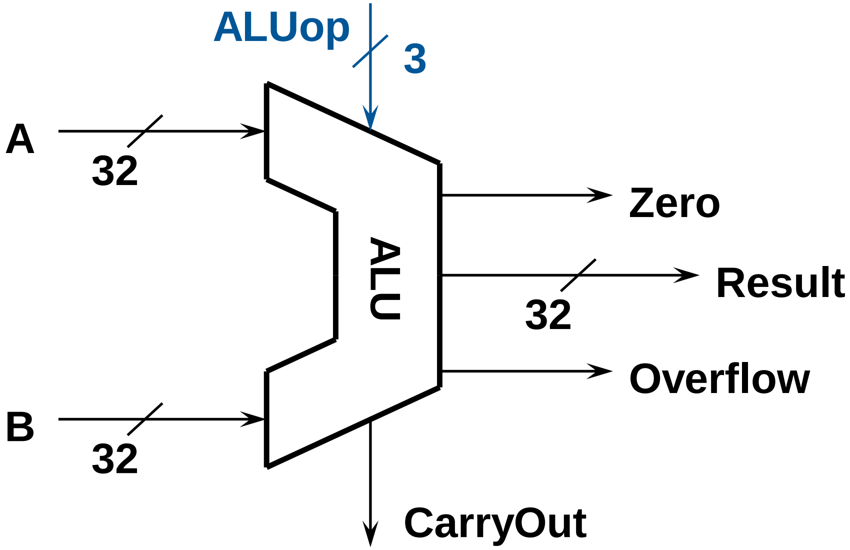




Note that the ‘overflow’ and ‘zero’ signals are not shown in this material, for simplicity. Basically, you need to design additional and specific circuits to output them. Take them as exercises with you.

A bit-slice ALU

- Design trick 2: Work on what you already have
  - Solve part of the problem and extend based on it.
  - E.g., realize the ‘and’, ‘or’, and ‘add’ functionalities, then extend to ‘sub’, and ‘slt’.



ALU Control (ALUop)	Operation
000	and
001	or
010	add
110	subtract
111	set-less-than

ALU Design

Designing the ALU

- Let's make a pre-design analysis.
- What are the digital building blocks really needed to realize the five operations ('and', 'or', 'add', 'sub', and 'slt')?
  - One AND gate, one OR gate, and one ADDER.
  - 'sub' can be done by the ADDER.
  - 'slt' needs to subtract, so it can also be done by the ADDER.
  - One multiplexer for the ALUop control signal to select and output one result from the multiple incoming results (namely, from AND, OR, ADDER)
  - Maybe more.

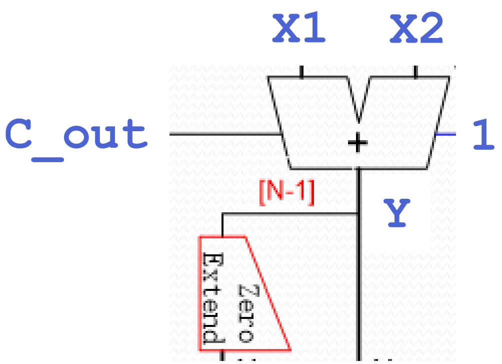
ALU Control (ALUop)	Operation
000	and
001	or
010	add
110	subtract
111	set-less-than

Designing the ALU

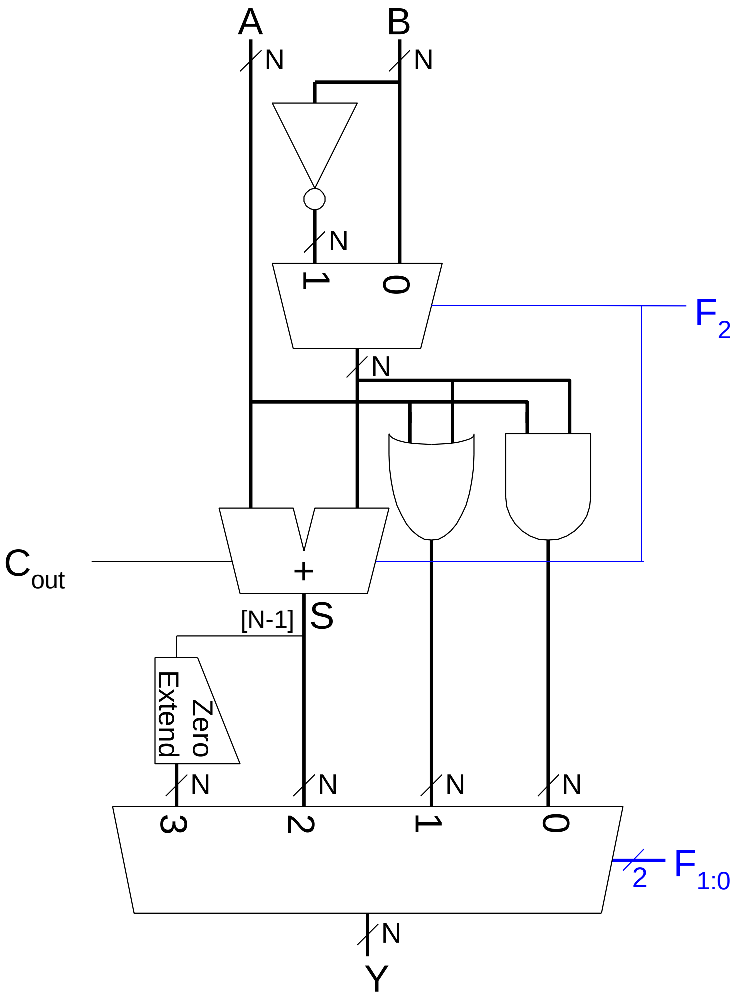
- Let's design the 'slt' functionality.
- What does 'slt' do, and what does it output?
  - Firstly, '`slt Y, X1, X2`' firstly subtracts `X2` from `X1` (i.e., `X1-X2`)
  - Then, it should compare the result with zero.
    - If `x1 – x2 < 0`, then `Y` is set (`Y = 1`)
    - Else, then `Y = 0`.
  - Given `Y` being 32-bit binary
    - If `x1 – x2 < 0`, then `Y` is set (`Y = 0000...0001`)
    - Else, then `Y = 0000...0000`.

But how to make `Y = 1` when `x1-x2<0`?

- In 2's complement format, the result of `x1-x2` is `1xxx...xxxx`, if `x1-x2 < 0`. Only need to shift the leading digit back, from position 31 to position 0. This is how `Y` is obtained.



The ALU Design

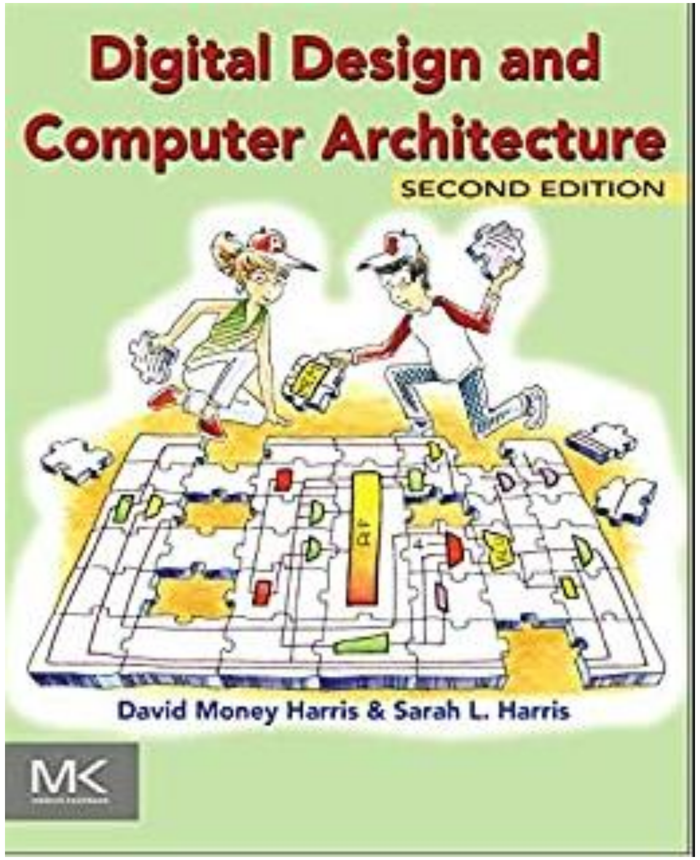


$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
110	A - B
111	SLT

Reference Textbook

Digital design and computer architecture, 1st or 2nd ed; by D. Harris & S. Harris

- ✓ Available in UNNC library (link)
- ✓ A simplified version of the Patterson & Hennessey Book.
- ✓ An ARM RISC ISA version also available in library.



University of Nottingham
UK | CHINA | MALAYSIA

Stay Tuned.