CS 250: Computer Architecture

Final Exam

Spring 2025

Benjamin Lobos Lertpunyaroj

May 8th, 10:30AM – 12:30PM

Exam contents and details for referencing

- Final exam is held in Fowler Hall on May 8th (Thursday), from 10:30 AM to 12:30 PM.
- Previous cumulative book chapters
  - Chapter 1 sections 1, 2, and 3.
  - Chapter 2, sections 1, 2, 3, 4, 5, 6, and 7.
  - Chapter 3, sections 1, 2, and 5.
  - Chapter 4, sections 1, 2, 3, 4, 5, 6, 7, and 8.
  - Chapter 5, sections 1, 2, 3, 4, 7, and 8.
  - Chapter 8 (Appendix A), sections 1, 2, 3 (but not PLAs or ROMs), 5, 7 (lightly), and 8.
- All lecture notes and lecture slides.
- All labs (1 - 11).
- The order of appearance of contents in this document is arbitrary.

Appendix A

Logic & Gates

An *asserted* signal is logically true, the *deasserted* is the opposite.

Two types of logic systems

Combinational logic: No memory in components, hence same output given same input.

Sequential logic: Memory in components, hence output depends on input and current memory state.

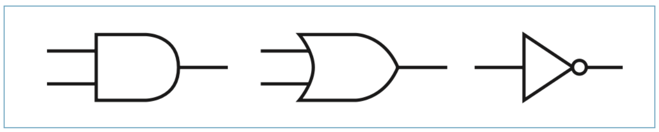


Figure 1: `AND` *gate,* `OR` *gate, and inverter*

The gates can be combined to form different forms of logic. An example of this is $\overline{\overline{A} + B}$ which is equivalent to $A \cdot \overline{B}$ by De Morgan's law, seen in Figure 2.

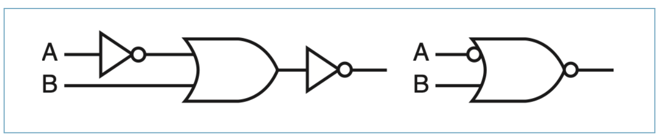
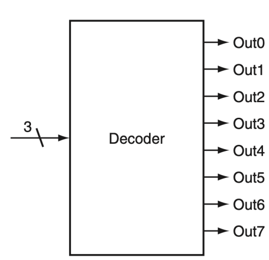


Figure 2: *Logic gate implementation of example formula*

Decoders & Multiplexors

A **decoder** is a logic block that has an n-bit input and 2^n outputs, where there is one unique true bit as output from a unique set of bytes of input.



a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

Figure 3: *3-bit input decoder that generates* $2^3 = 8$ *different outputs (Out0 – Out7)*

$$2^n \text{ outputs } \therefore \log_2(\text{output}) = \text{input bits}$$

Encoders are the other way around.

Multiplexors have a selector input (or control value), that will determine which inputs will become outputs.

In the case of the two-input MUX, its representation is the following, $C = (A \cdot \overline{S}) + (B \cdot S)$, using n (data inputs) AND gates, and one OR gate.

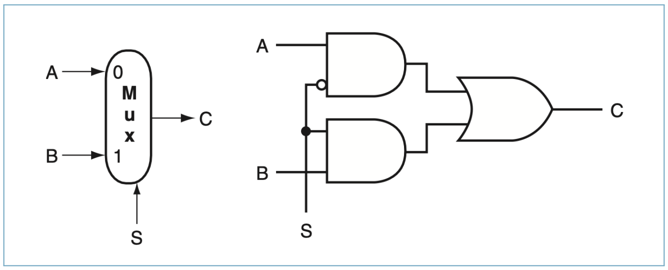


Figure 4: *Two-input multiplexor that generates one output depending on the selector input S*

$$n \text{ (data inputs)} \therefore \log_2 n = S \text{ selector bits required to represent all inputs}$$

Often times a decoder generates n bits for a MUX, to be used as a selector signal.

Buses

A collection of data lines that is treated as a single logical signal.

When showing a logic unit whose inputs and outputs are buses, the unit must be replicated a sufficient number of times to accommodate the width of the input.

You can use multiplexors to select between two buses, requiring n inputs to represent n-bit buses.

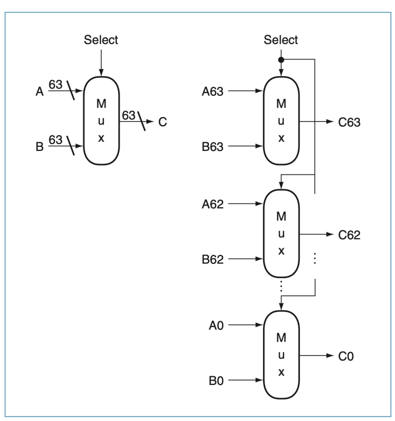


Figure 5: *1-bit multiplexors replicated 64 times to represent two 64-bit buses*

ALUs

Operation done by the ALU

Logic operations: `AND` and `OR` gate operations, with `NOR` being available through an inversion of both input signals with `AInvert` and `BInvert` control signals.

Arithmetic operations: Addition and subtraction through the full adder, and `BInvert` control signal on one input for determining the type of operation.

The LEGv8 word is 64 bits wide, as such a 64 bit wide ALU is required (64 1-bit ALUs).
In its simplest form, a 1-bit logical unit for `AND` and `OR` operations simply requires a multiplexor an a one bit control signal to select between the two operations ($2^1 = 1$).

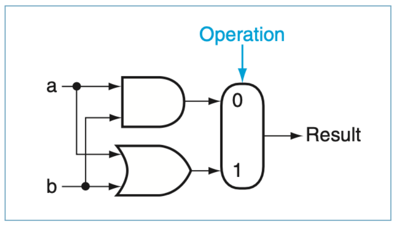


Figure 6: *1-bit logical unit for* `AND` *and* `OR` *operations*

Implementing addition requires two input operands, one output, a `CarryIn` bit carried from the less-significant bits of the operation (i.e. another 1-bit logical unit), and a `CarryOut` bit to be carried forward to the next more significant bit.

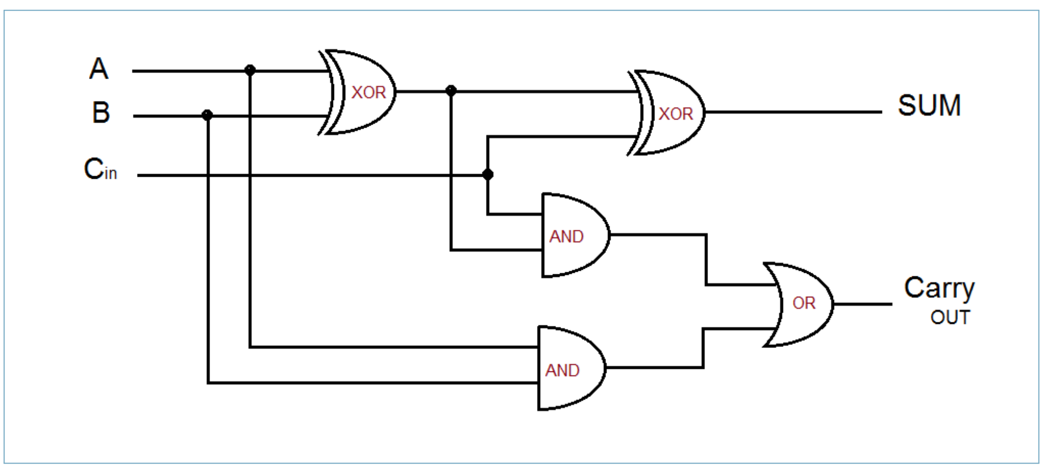


Figure 7: *Full adder that performs mod 2 addition*

The combination of the adder and the logic gates, coupled with a multiplexor with a control signal to determine the operation makes a complete 1-bit ALU, which can be seen in Figure 8.

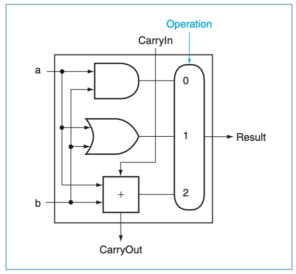


Figure 8: *1-bit alu with logical operations and addition*

For expanding to a 64-bit ALU, the adders have to set up a ripple carry from the least to the most significant bit.

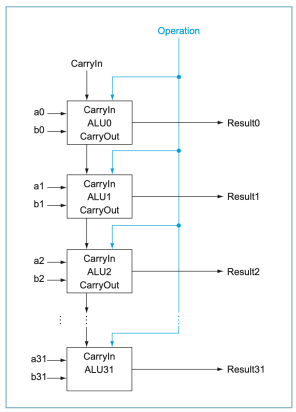


Figure 9: *Ripple carry implemented for a 64-bit ALU*

By inverting the second input (`BInvert` = 1, seen in Figure 10) and setting `CarryIn` to 1 in the least significant bit of the ALU, we get two's complement subtraction of `b` from `a`.
To implement a `NOR` function, existing components can be combined, $(\overline{a+b}) = \overline{a} \cdot \overline{b}$ (DeMorgan's theorem), which means we need an `AND` and two inverters for both `a` and `b`, seen in Figure 10

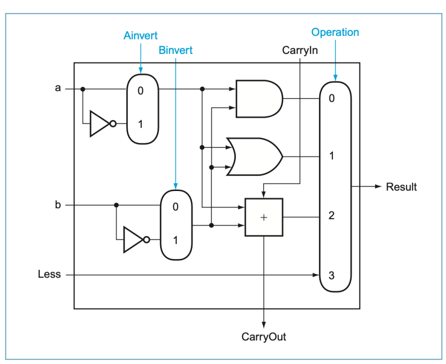


Figure 10: *1-bit ALU that performs subtraction, and* `NOR` *operations*

On a 64-bit ALU we can use a zero flag to help with conditional branch instructions in LEGv8 (e.g.

`CBZ`), as they receive to inputs and require to test if the subtraction has a zero.

The following represents this with an inversion of an `OR` tree on all results from the subtraction considering a 64-bit subtraction, fully represented in Figure 11.

$$Zero = \overline{(R_0 + R_1 + R_2 + ... + R_{63})}$$

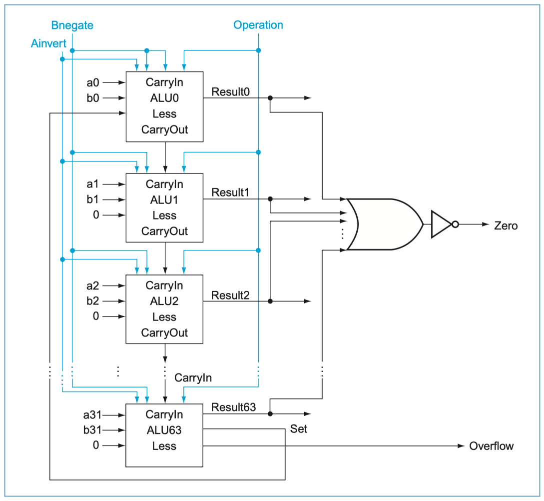


Figure 11: *64-bit ALU* `OR` *tree and an inverter for determining the Zero flag*

For a generalized symbol of the ALU, Figure 12, where `ALU operation` is the control signal of the MUX that determines the type of operation.

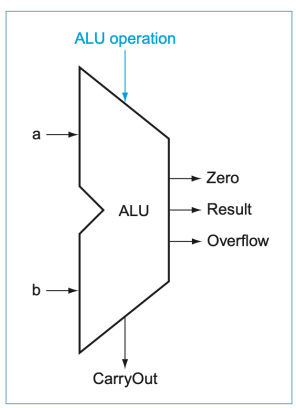


Figure 12: *General symbol for an ALU or an adder*

Clocks

Clocking methodology semantics

Edge triggered clocking: State changes occur on a clock edge.

Synchronous system: Type of memory system where data is read only when a clock signal indicates stability (i.e. non-changing value).

A combinational logic block, recieves an input and then generates an output for a state element which is updated on a clock edge.

An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race condition.

For this to work, the clock cycle must be long enough for the state element to have received a stable input before the next active clock edge.

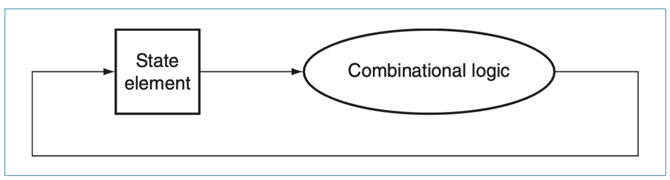


Figure 13: *Edge-triggered state element to be read and written to in one active clock edge*

One such state element is the register file.

Flip-flops & Latches

Types of clocked memory elements

Flip-flops: Edge-triggered element that changes the stored state only at a clock edge.

Latches: Level-sensitive element that changes the stored state at any time the clock is asserted.

Flip-flops are build upon latches and are going to be used in edge-triggered systems.

A **D flip-flop** or **D latch** is used for storing the value of one data input signal, in the internal memory, at the clock edge.

To implement a **D latch**, it requires two inputs, the data to be stored `D`, and the clock signal `C`, producing two outputs, the value of the internal state `Q`, and its complement $\overline{\texttt{Q}}$.

The implementation has cross-coupled `NOR` gates that store the state value unless `C` is asserted, in which case `D` replaces the value of `Q` and is stored.

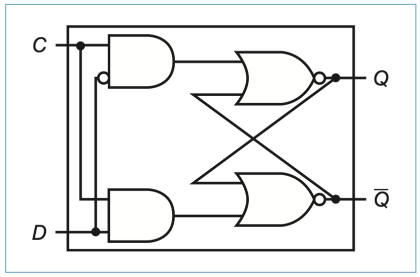


Figure 14: *D latch, composed of crossed* `NOR` *gates and a* `SR` *latch*

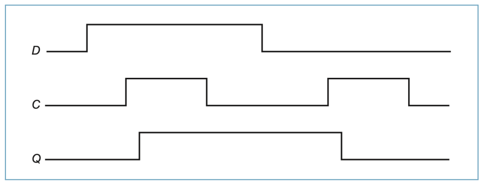


Figure 15: *Progression of a D latch, assuming output is initially deasserted*

To implement a **D flip-flop**, with a falling-edge trigger, we can use two D latches, master and slave. Master sets input `D` when `C` is asserted. When `C` falls, master is closed, but slave is open and gets its input from master's `Q`.

In this sense, the rising-edge represents when the master takes in the `D` value, and the falling-edge represents when the slave takes in the master's `D` producing the final `Q`.

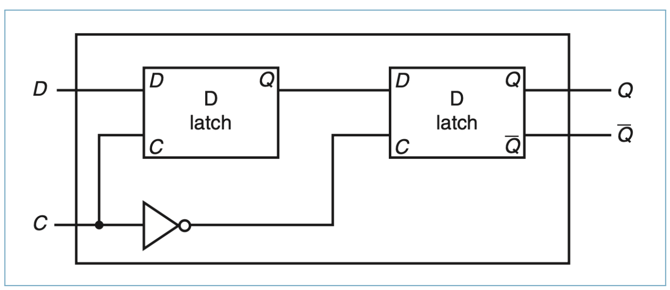


Figure 16: *D flip-flop with a falling-edge trigger made from two D latches, master and slave*

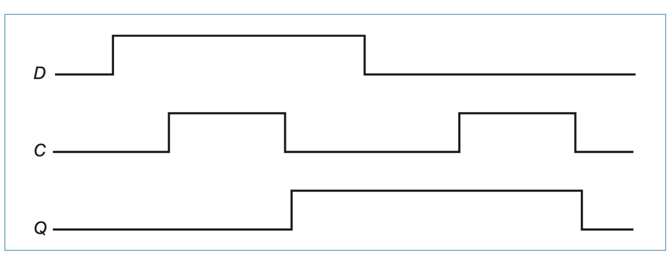


Figure 17: *Progression of a D flip-flop with a falling-edge trigger, where output is initially deasserted*

The minimum time `D` must retain a valid input is the setup time plus the hold time (after edge).

Register files

A **register file** consists of a bunch of **registers** that can be read and written to, and a `WriteReg` control signal (clock).

For writing it requires the control signal, the number of the register to write to (`Write register`), and the data to write (`Write data`).

For reading it requires the numbers of the registers to read from (`Read register number 1 & 2`), and it outputs the read contents from two registers (`Read data 1 & 2`).

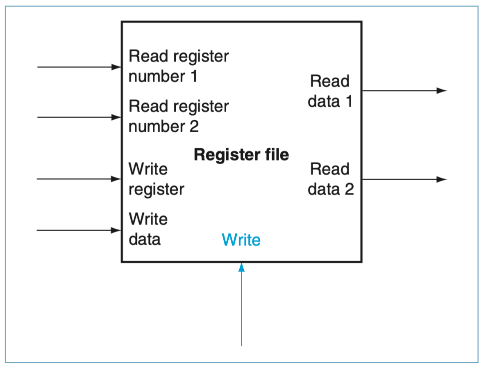


Figure 18: *Register file with two read ports and one write port*

The implementation for the write port consists of a decoder that will select one of the `n - 1` registers that will be `AND`ed with the `WriteReg` signal to act as the `C` input for the registers (D flip-flops). The `D` input for every register is the `Write data` input from the reg. file.

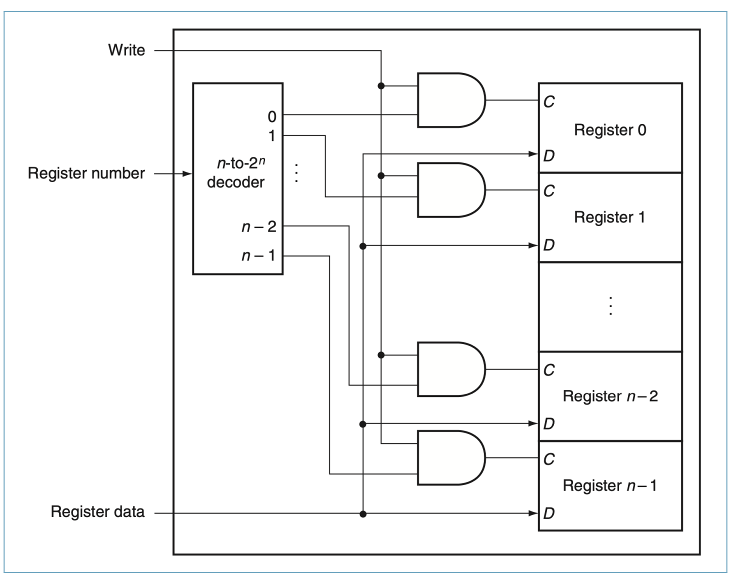


Figure 19: *Write implementation in the register file*

The implementation for the two read ports consists of using the stored state of the registers (`Q` output), as inputs for two different MUXes that use the `Read register number 1 & 2` reg. file inputs as control signals to output the information of the two registers specified.

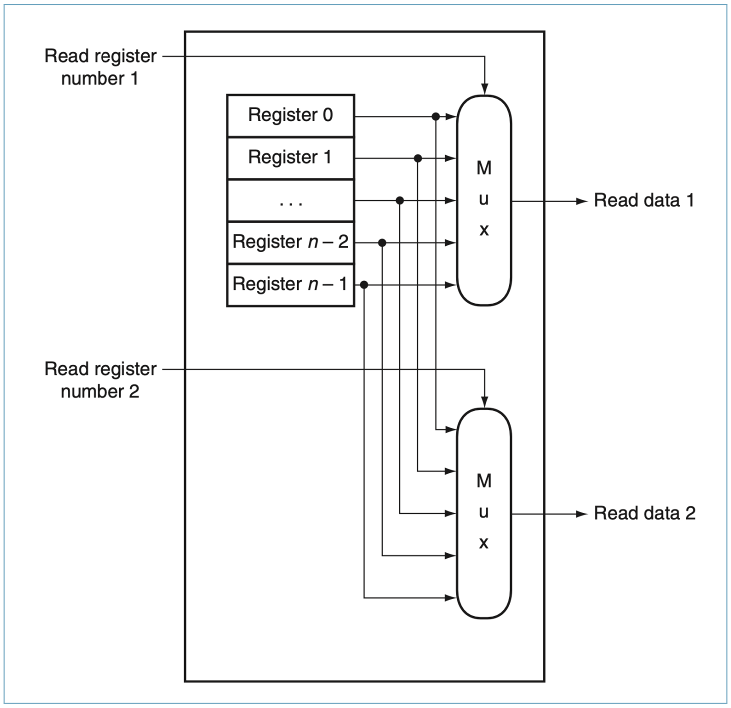


Figure 20: *Read implementation in the register file*