Reminder

- **Coursework 1: due Wed @ 4pm**

- **Do:**
  - Have correct code
    - Compiles/builds/runs without warnings or errors
    - MIPS & C syntax & semantics are followed
  - Have well-structured code
    - Use functions; no goto’s
  - Have readable code
    - Meaningful comments
    - Meaningful names for functions, labels, C variables, etc.
Reminder

- **Coursework 1: due next Thur @ 4pm**

- **Don’t:**
  - Be late!
  - Ask me for extensions
    - ITO and UG2 year organizer (Dr. Rik Sarkar) handle these
  - Plagiarize!

I WILL NOT PLAGIARIZE ANOTHER'S WORK
I WILL NOT PLAGIARIZE ANOTHER'S WORK
I WILL NOT PLAGIARIZE ANOTHER'S WORK
I WILL NOT PLAGIARIZE ANOTHER'S WORK
I WILL NOT PLAGIARIZE ANOTHER'S WORK
I WILL NOT PLAGIARIZE
Logic design overview

**Binary digital logic circuits:**

- Two voltage levels (ground and supply voltage) for 0 and 1
  - Built from transistors used as on/off switches
  - Analog circuits not very suitable for generic computing
  - Digital logic with more than two states is not practical

**Combinational logic:** output depends only on the current inputs (no memory of past inputs)

![Combinational Logic Diagram](image)

**Sequential logic:** output depends on the current inputs as well as (some) previous inputs → requires “memory”
Combinational logic circuits

- Inverter (or NOT gate): 1 input and 1 output
  “invert the input signal”

```
input → output
```

<table>
<thead>
<tr>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \text{OUT} = \overline{\text{IN}} \]

Truth table
Combinational logic circuits

- **Inverter (or NOT gate):** 1 input and 1 output
  “invert the input signal”

```
IN  |  OUT
---|---
0   |  1
1   |  0
OUT = \overline{IN}
```

- **AND gate:** 2 inputs and 1 output
  “output 1 only if both inputs are 1”

```
IN_1 | IN_2 | OUT
-----|-----|-----
0    |  0  |  0  
0    |  1  |  0  
1    |  0  |  0  
1    |  1  |  1  
OUT = \overline{IN_1} \cdot \overline{IN_2}
```
Combinational logic circuits

- OR gate: “output 1 if at least one input is 1”

<table>
<thead>
<tr>
<th>IN1</th>
<th>IN2</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \text{OUT} = \text{IN}_1 + \text{IN}_2 \]
Combinational logic circuits

- **OR gate:** “output 1 if at least one input is 1”

  \[
  \begin{array}{c|c|c}
  \text{IN}_1 & \text{IN}_2 & \text{OUT} \\
  \hline
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 1 \\
  \end{array}
  \]

  \(\text{OUT} = \text{IN}_1 + \text{IN}_2\)

- **NOR gate:** “output 1 if no input is 1” (NOT OR)

  \[
  \begin{array}{c|c|c}
  \text{IN}_1 & \text{IN}_2 & \text{OUT} \\
  \hline
  0 & 0 & 1 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 0 \\
  \end{array}
  \]

  \(\text{OUT} = \overline{\text{IN}_1 + \text{IN}_2}\)
Combinational logic circuits

- **AND gate:** “output 1 if both inputs are 1”

  ![AND gate diagram]

<table>
<thead>
<tr>
<th>IN₁</th>
<th>IN₂</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

  \[ \text{OUT} = \text{IN}_1 \cdot \text{IN}_2 \]

- **NAND gate:** “output 1 if both inputs are not 1” (NOT AND)

  ![NAND gate diagram]

<table>
<thead>
<tr>
<th>IN₁</th>
<th>IN₂</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

  \[ \text{OUT} = \overline{\text{IN}_1 \cdot \text{IN}_2} \]
Combinational logic circuits

- Multiple-input gates:

\[ \text{AND} \]
\[ \begin{array}{c}
\text{IN}_1 \\
\vdots \\
\text{IN}_n \\
\end{array} \quad \text{OUT} \]
\[ \text{OUT} = 1 \text{ if all } \text{IN}_i = 1 \]

\[ \text{OR} \]
\[ \begin{array}{c}
\text{IN}_1 \\
\vdots \\
\text{IN}_n \\
\end{array} \quad \text{OUT} \]
\[ \text{OUT} = 1 \text{ if any } \text{IN}_i = 1 \]
Combinational logic circuits

- **Functional completeness:**
  - Set of gates that is sufficient to express any boolean function

- **Examples:**
  - AND + OR + NOT
  - NAND
  - NOR
Multiplexer (mux)

- Multiplexer: a circuit for selecting one of multiple inputs

\[ z = \begin{cases} 
  i_0, & \text{if } c=0 \\
  i_1, & \text{if } c=1 
\end{cases} \]

\[
\begin{array}{c|cc|c}
 c & i_0 & i_1 & z \\
\hline
 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 \\
 0 & 1 & 0 & 1 \\
 0 & 1 & 1 & 1 \\
 1 & 0 & 0 & 0 \\
 1 & 0 & 1 & 1 \\
 1 & 1 & 0 & 0 \\
 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
z = \overline{c}.i_0.\overline{i_1} + \overline{c}.i_0.i_1 + c.\overline{i_0}.i_1 + c.i_0.i_1
\]

\[
= \overline{c}.i_0.(\overline{i_1} + i_1) + c.(\overline{i_0} + i_0).i_1
\]

\[
= \overline{c}.i_0 + c.i_1 \quad \text{minimized}
\]

“sum of products form”
A multiplexer implementation

- Sum of products form: $i_1.c + i_0.\overline{c}$
  - Can be implemented with 1 inverter, 2 AND gates & 1 OR gate:

- Sum of products is not practical for circuits with large number of inputs ($n$)
  - The number of possible products can be proportional to $2^n$
Arithmetic circuits

- 32-bit adder

- Idea: modularize!
  - Design a generic 1-bit adder block
  - Replicate it N number of times for an N-bit adder

64 inputs → too complex for sum of products
Arithmetic circuits

- 32-bit adder

[Diagram of a 32-bit adder with inputs a₀, a₃₁, b₀, b₃₁, and outputs s₀, s₃₂.]

64 inputs → too complex for sum of products

- Full adder:

[Diagram of a full adder with inputs a, b, and c (carry-in), and outputs sum and carry-out.]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>carry</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

sum = a.b.c + a.b.c + a.b.c + a.b.c

carry = b.c + a.c + a.b
Ripple carry adder

- 32-bit adder: chain of 32 full adders

- Carry bits \( c_i \) are computed in sequence \( c_1, c_2, \ldots, c_{32} \) (where \( c_{32} = s_{32} \)), as \( c_i \) depends on \( c_{i-1} \)

- Since sum bits \( s_i \) also depend on \( c_i \), they too are computed in sequence
Propagation delays

- Propagation delay = time delay between input signal change and output signal change at the other end

- Delay depends on:
  1. technology (transistor parameters, wire capacitance, etc.)
  2. delay through each gate (function of gate type)
  3. number of gates driven by a gate’s output (fan out)

- e.g.: 2-input mux: NOT $\rightarrow$ AND $\rightarrow$ OR $\Rightarrow$ 3 gate delays. Fast!

- What’s the delay of a 32-bit ripple carry adder?
  - 65 gate delays $\Rightarrow$ slow
  - AND2 + OR3 for each of 31 carries to propagate;
    followed by NOT + AND3 + OR4 for $S_{31}$
Practice problem:

Design a circuit that, given a 4-bit unsigned input, outputs:

1 - if the input value is $\geq 7$
0 - otherwise

What is the propagation delay of the circuit?
What is the delay if only 1- and 2-input gates are allowed?
Sequential logic circuits

- Output depends on current AND past inputs
  - The circuit has memory
- Sequences of inputs generate sequences of outputs ⇒ **sequential logic**
  - With $n$ feedback signals → up to $2^n$ stable states
SR Latch: the basic sequential circuit

SR latch
– Inputs: R, S
– Feedback: q, q
– Output: Q
SR Latch
SR Latch

- **Truth table:**
  
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>$Q_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$Q_{i-1}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$inv$</td>
</tr>
</tbody>
</table>

  inv=invalid

- **Usage: 1-bit memory**
  
  - Keep the value in memory by maintaining $S=0$ and $R=0$
  - Set the value in memory to 0 (or 1) by setting $R=1$ (or $S=1$) for a short time

  ![SR Latch Diagram]

  ![SR Latch Truth Table]

Timing of events

- Asynchronous sequential logic
  - State (and possibly output) of circuit changes whenever inputs change

- Synchronous sequential logic
  - State (and possibly output) can only change at times synchronized to an external signal → the clock
Using clock to build a memory element

- **Level-triggered D latch:** whenever clock is 1, D is propagated to Q
Using clock to build a memory element

- **Level-triggered D latch:** whenever clock is 1, D is propagated to Q

- **Edge-triggered D flip-flop:** on a positive clock edge, D is propagated to Q
Register

- Tie multiple D flip-flops together using a common clock
- E.g., 4-bit register:

```
D3  D2  D1  D0  
Q3  Q2  Q1  Q0  
```

Clock
General sequential logic circuit

- **Operation:**
  - At every rising clock edge next state signals are propagated to current state signals
  - Current state signals plus inputs work through combinational logic and generate output and next state signals
Hardware FSM

- A sequential circuit is a (deterministic) Finite State Machine – FSM

- Example: Vending machine
  - Accepts 10p, 20p coins, sells one product costing 30p, no change given
  - Coin reader has 2 signals: $a$, $b$ for 10p, 20p coins respectively. These are the inputs to our FSM
  - Output $z$ asserted when 30p or more has been paid in
FSM implementation

- Methodology:
  - Choose encoding for states, e.g. S0=00, …, S3=11
  - Build truth table for the next state $s_1'$, $s_0'$ and output $z$
  - Generate logic equations for $s_1'$, $s_0'$, $z$
  - Design comb logic from logic equations and add state-holding register

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$s_0$</th>
<th>$a$</th>
<th>$b$</th>
<th>$s_1'$</th>
<th>$s_0'$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>