EECS151/251A
Spring 2020
Digital Design and
Integrated Circuits

Instructor:
John Wawrzynek

Lecture 21:
Wrap-up
Outline

- Important takeaways
- Exam Topics
Why Study and Learn Digital Design?

- We expect that many of our graduates will eventually be employed as designers.
  - *Digital design is not a spectator sport.* The only way to learn it, and to appreciate the issues, is to do it.
  - To a large extent, it comes with practice/experience (this course is just the beginning).
  - Another way to get better is to study other designs. Not time to do much of this during the semester, but a good practice for later.

- However, a significant percentage of our graduates will not be digital designers. What’s in it for them?
  - Better manager of designers, marketers, field engineers, etc.
  - Better researcher/scientist/designer in related areas
    – Software engineers, fabrication process development, etc.
  - To become a better user of electronic systems.
In What Context Will You be Designing?

Engineers learn so that they can build. Scientists build so that they can learn.

- Electronic design is a critical tool for most areas of pure science:
  - Astrophysics – special electronics used for processing radio antenna signals.
  - Genomics – special processing architectures for DNA string matching.
  - In general - sensor processing, control, and number crunching.
  - Machine Learning now relies heavily on special hardware.
  - In some fields, computation has replaced experimentation – particle physics, world weather prediction (fluid dynamics).

- In computer engineering, prototypes often designed, implemented, and studied to “prove out” an idea. Common within universities and industrial research labs. Lessons learned and proven ideas often transferred to industry through licensing, technical communications, or startup companies.
  - RISC processors were first proved out at Berkeley and IBM Research
Of course, companies are the primary employer of designers. Provide some useful products to society or government and make a profit for the shareholders.

Interesting recent shift
- All software giants now have hardware design teams (embedded and chips)
- Google, Amazon, Facebook, Microsoft, ...
Ten Big Ideas from EECS151

1. **Modularity and Hierarchy** is an important way to describe and think about digital systems.

2. **Parallelism** is a key property of hardware systems and distinguishes them from serial software execution.

3. **Clocking** and the use of state elements (latches, flip-flops, and memories) control the flow of data.

4. **Cost/Performance/Power tradeoffs** are possible at all levels of the system design.

5. **Boolean Algebra** and other logic representations.

6. **Hardware Description Languages (HDLs)** and **Logic Synthesis** are a central tool for digital design.

7. **Datapath + Controller** is an effective design pattern.

8. **Finite State Machines** abstraction gives us a way to model any digital system – used for designing controllers.

9. **Arithmetic circuits** are often based on “long-hand” arithmetic techniques.

10. **FPGAs + ASICs** give us a convenient and flexible implementation technology.
What We Didn’t Cover

- Design Verification and Testing
  - Industrial designers spend more than half their time testing and verifying correctness of their designs.
    - Some of this covered in the lab and a bit in lecture. Didn’t cover rigorous testing procedures.
  - Most industrial products are designed from the start for testability. Important for design verification and later for manufacturing test.
  - Related: Fault modeling and fault tolerant design.

- Other High-level Optimization Techniques
  - High-level Synthesis - now starting to catch on

- Other High-level Architectures: GPUs, video processing, network routers, …

- Asynchronous Design
Most Closely Related Courses

- **CS152 Computer Architecture and Engineering**
  - Design and Analysis of Microprocessors
  - Applies basic design concepts from EECS151

- **EE241B Digital Integrated Circuits**
  - Transistor-level design of ICs
  - More on Advanced ASIC Tool use

- **CS250 VLSI Systems Design**
  - Advanced-undergrad/grad course
  - Design tradeoffs at the chip design level
Future Design Issues

- Automatic High-level synthesis (HLS) and optimization (with micro-architecture synthesis) and hardware/software co-design.

- Current trend is towards “system on a chip” (SOC) design methodology:
  - Pre-designed subsystems (processor cores, bus controllers, memory systems, network interfaces, etc.) connected with standard on-chip interconnect or bus.
  - Strong emphasis on “accelerators”.

- A number of alternatives to silicon VLSI have been proposed, including techniques based on:
  - Carbon nanotubes*, molecular electronics, quantum mechanics, and biological processes.
  - How will these change the way we design systems?

*In 2012, IBM produced a sub-10 nm carbon nanotube transistor that outperformed silicon on speed and power. "The superior low-voltage performance of the sub-10 nm CNT transistor proves the viability of nanotubes for consideration in future aggressively scaled transistor technologies", according to the abstract of the paper in Nano Letters.
Final Exam and Project Info

- Exam held this Thursday 7-10PM.
  - “Comprehensive” Final Exam
  - Emphasis on second half, but some coverage of first half
  - Take at home format, Q/A through piazza
- Project interviews (end of RRR week)
- Project final reports (beginning of Exam week)
The exam will be a “take home exam” and take place Thursday April 30, 7-10PM. The exam comprises a set of questions with 1 point per expected minute of completion with a total of approximately 90 points. 251A students will be asked to complete extra questions. All students are allowed to refer to your notes, the class lecture notes, and any other reference materials that you have available. However, the problems are challenging and if you are not suitable familiar with the course topics, you may not have much time to look at notes. You are not allowed to speak with anyone on any topic related to the course during the exam period. After completing the exam, you will be asked to sign a statement attesting that you did not discuss the exam problems with anyone else. You will turn in your answers with Gradescope as you do your homework.

As with the first exam, clarification questions can be directed to the teaching staff via piazza.

Topics:
The final exam will be comprehensive and test all topics covered this semester. However, emphasis will be placed on topics covered after the midterm exam—those listed below.

1. How to Design a RISC-V Single-Cycle Processor from the ISA
2. Processor Pipelining Hazards and Mechanisms
3. Sources of Power and Energy consumption in Digital ICs
4. Principles Behind Six Low-power Design Techniques
5. How to Improve Energy Efficiency through Parallelism and Pipelining
6. Memory Block Internal Architecture
7. SRAM Cell and Read/Write Operation
8. Memory Block Periphery Circuits
9. Memory Decoder Design
10. DRAM Cell and Read/Write Operation
11. Dual-port Memory Architecture
12. Cascading Memory blocks for More Width, Depth, and Ports
13. FIFO Implementation
14. Serialization versus Parallelization in Iterative Computations
15. Principles of Pipelining and Restrictions of Loops
16. C-Slow Technique for Pipelining Loops
17. List Processor Design and Optimizations
18. Modulo Scheduling
19. Carry Select Adder Design
20. Carry Lookahead and Parallel Prefix Adders
21. Bit-Serial Addition
22. Array Multiplier Design
23. Carry Save Addition
24. Signed Multiplication
25. Bit-Serial Multiplication
26. CSD Multiplier Design
27. Booth Encoding (251A only)
28. Log and Barrel Shifters Design and Analysis
29. Use of Counters in Controller Design
30. Binary Counter Design and Optimization
31. Ring Counter Design
32. Effect of Clock Uncertainties on Maximum Clock Frequency
33. Source of Clock Uncertainties
34. Principle of Good Clock Distribution
35. IR and dI/dt effects in Power distribution
36. Types and Sources of Faults in ICs
37. Hamming Codes
Single-Cycle RISC-V RV32I Datapath
Pipelined Processor
Total Power = \( P_{\text{switching}} + P_{\text{short-circuit}} + P_{\text{leakage}} \)

\[ I_{DSub} = k \cdot e^{\frac{-q \cdot V_T}{a \cdot k_B \cdot T}} \]
Six low-power design techniques

- Parallelism and pipelining
- Power-down idle transistors
- Slow down non-critical paths
- Clock gating
- Data-dependent processing
- Thermal management
Gate delay roughly linear with Vdd

Block processes stereo audio. 1/2 of clocks for "left", 1/2 for "right".

Top block processes "left", bottom "right".

This magic trick brought to you by Cory Hall...
Memory Architecture Overview

- **Word lines** used to select a row for reading or writing
- **Bit lines** carry data to/from periphery
- **Core aspect ratio** keep close to 1 to help balance delay on word line versus bit line
- **Address bits** are divided between the two decoders
- **Row decoder** used to select word line
- **Column decoder** used to select one or more columns for input/output of data
SRAM read/write operations
Periphery

- Decoders
- Sense Amplifiers
- Input/Output Buffers
- Control / Timing Circuitry
Row Decoder

- Expands L-K address lines into $2^{L-K}$ word lines

Example: decoder for 8Kx8 memory block

- core arranged as 256x256 cells
- Need 256 AND gates, each driving one word line
1-Transistor DRAM Cell

Write: $C_S$ is charged or discharged by asserting WL and BL.

Read: Charge redistribution takes places between bit line and storage capacitance

$C_S << C_{BL}$ Voltage swing is small; typically around 250 mV.

- To get sufficient $C_S$, special IC process is used
- Cell reading is destructive, therefore read operation always is followed by a write-back
- Cell looses charge (leaks away in ms - highly temperature dependent), therefore cells occasionally need to be “refreshed” - read/write cycle
Add decoder, another set of read/write logic, bits lines, word lines:

- Example cell: SRAM
- Repeat everything but cross-coupled inverters.
- This scheme extends up to a couple more ports, then need to add additional transistors.
Cascading Memory-Blocks

How to make larger memory blocks out of smaller ones.

Increasing the depth. Example: given 1Kx8, want 2Kx8
FIFO Implementation Details

• Assume, dual-port memory with asynchronous read, synchronous write.
• Binary counter for each of read and write address. CEs (count enable) controlled by WE and RE.
• Equal comparator to see when pointers match.
• State elements for FULL and EMPTY flags:

<table>
<thead>
<tr>
<th>WE</th>
<th>RE</th>
<th>EMPTY _i</th>
<th>FULL _i</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 1</td>
<td>EMPTY _i-1</td>
<td>FULL _i-1</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>EMPTY _i-1</td>
<td>FULL _i-1</td>
</tr>
</tbody>
</table>

• Control logic (FSM) with truth-table (draft) shown to left.

*Actually need 2 signals: “will be equal after read” and “will be equal after write”
Time-Multiplexing

• *Time multiplex* single ALU for all adds and multiplies:

• Attempts to minimize cost at the expense of time.
  – Need to add extra register, muxes, control.

• If we adopt above approach, we can then consider the combinational hardware circuit diagram as an *abstract computation-graph*.

  ![Diagram](chart]

  Using other primitives, other coverings are possible.

• This time-multiplexing “covers” the computation graph by performing the action of each node one at a time. (Sort of *emulates* it.)
**Limits on Pipelining**

- Without FF overhead, throughput improvement $\alpha$ # of stages.
- After many stages are added FF overhead begins to dominate:

  - clock skew contributes to clock overhead
  - unequal stages
  - FFs dominate cost
  - clock distribution power consumption
  - feedback (dependencies between loop iterations)

**Figure:**
- The graph illustrates the throughput ($1/T$) in real and ideal scenarios as a function of the number of stages.
- FF “overhead” is the setup and clk to Q times.
- Half the clock period in FF overhead.
Pipelining Loops with Feedback

“Loop carry dependency”

However, we can overlap the “non-feedback” part of the iterations:

Add is associative and commutative. Therefore we can reorder the computation to shorten the delay of the feedback path:

\[ y_i = (y_{i-1} + x_i) + a = (a + x_i) + y_{i-1} \]

- Pipelining is limited to 2 stages.

“Shorten” the feedback path.
“C-slow” Technique

- Essentially this means we go ahead and cut feedback path:
- This makes operations in adjacent pipeline stages independent and allows full cycle for each:
- C computations (in this case C=2) can use the pipeline simultaneously.
- Must be independent.
- Input MUX interleaves input streams.
- Each stream runs at half the pipeline frequency.
- Pipeline achieves full throughput.

Multithreaded Processors use this.

<table>
<thead>
<tr>
<th>add₁</th>
<th>x+b</th>
<th>x+b</th>
<th>x+b</th>
<th>x+b</th>
<th>x+b</th>
<th>x+b</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>ay</td>
<td>ay</td>
<td>ay</td>
<td>ay</td>
<td>ay</td>
<td>ay</td>
</tr>
<tr>
<td>add₂</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>
5. Optimization, Architecture #4

- **Datapath:**

- **Incremental cost:**
  - Addition of another register & mux, adder mux, and control.

- **Performance:** find max time of the four actions
  1. $X \leftrightarrow \text{Memory[NUMA]}$, $\text{NUMA} \leftrightarrow \text{NEXT}+1$; $0.5+1+10+1+0.5 = 13\text{ns}$
  2. $\text{NEXT} \leftrightarrow \text{Memory[NEXT]}$, $\text{SUM} \leftrightarrow \text{SUM}+X$; same for all $\Rightarrow T > 13\text{ns}, F < 77\text{MHz}$
Modulo Scheduling List Processor

- Assuming a single adder and a single ported memory. Minimal schedule section length = 2.
  Because both memory and adder are used for 2 cycles during one iteration.

\[
\begin{align*}
\text{memory} & \quad \text{next}_i \\
\text{adder} & \quad \text{numa}_i \\
\text{memory} & \quad \text{next}_i \\
\text{adder} & \quad \text{numa}_i \\
\text{memory} & \quad \text{next}_i \\
\text{adder} & \quad \text{numa}_i
\end{align*}
\]

- Finished schedule for 4 iterations:

<table>
<thead>
<tr>
<th>Memory</th>
<th>next(_1)</th>
<th>next(_2)</th>
<th>x(_1)</th>
<th>next(_3)</th>
<th>x(_2)</th>
<th>next(_4)</th>
<th>x(_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td>numa(_1)</td>
<td>numa(_2)</td>
<td>sum(_1)</td>
<td>numa(_3)</td>
<td>sum(_2)</td>
<td>numa(_4)</td>
<td>sum(_3)</td>
</tr>
</tbody>
</table>
Carry Select Adder

- Extending Carry-select to multiple blocks

What is the optimal # of blocks and # of bits/block?
- If blocks too small delay dominated by total mux delay
- If blocks too large delay dominated by adder ripple delay

$T \propto \sqrt{N}$, Cost $\approx 2 \times \text{ripple} + \text{muxes}$
Parallel Prefix Adder Example

\[ G = g_3 + g_2 p_3 (g_1 + g_0 p_1) p_3 p_2 \]
\[ G = g_3 + g_2 p_3 + g_1 p_3 p_2 + g_0 p_3 p_2 p_1 \]
\[ G = c_3 \]

\[ G = g_3 + g_2 p_3 + g_1 p_2 + g_0 p_2 p_1 \]
\[ G = g_2 + g_1 p_2 + g_0 p_2 p_1 \]
\[ G = g_1 + g_0 p_1 \]

\[ P = p_3 p_2 \]
\[ P = p_2 p_1 \]
\[ P = p_1 p_0 \]

\[ s_i = a_i \oplus b_i \oplus c_i = p_i \oplus c_i \]
Bit-serial Adder

- Addition of 2 n-bit numbers:
  - takes n clock cycles,
  - uses 1 FF, 1 FA cell, plus registers
  - the bit streams may come from or go to other circuits, therefore the registers might not be needed.

- A, B, and R held in shift-registers. Shift right once per clock cycle.
- Reset is asserted by controller.
**Combinational Multiplier (unsigned)**

\[
\begin{array}{cccccc}
X_3 & X_2 & X_1 & X_0 & \text{multiplicand} & \rightarrow \\
* & Y_3 & Y_2 & Y_1 & Y_0 & \text{multiplier} \\
\hline
\end{array}
\]

\[
\begin{array}{cccccc}
X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 & \text{Partial products, one for each bit in multiplier (each bit needs just one AND gate)} \\
+ & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 & \\
+ & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 & \\
+ & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 & \\
\hline
Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0 \\
\end{array}
\]

Propagation delay \(\sim2N\)
2’s Complement Multiplication

\[
\begin{array}{cccc}
\overline{x}_3 \overline{y}_0 & x_2 y_0 & x_1 y_0 & x_0 y_0 \\
+ & \overline{x}_3 y_1 & x_2 y_1 & x_1 y_1 & x_0 y_1 \\
+ & x_3 y_2 & x_2 y_2 & x_1 y_2 & x_0 y_2 \\
+ & 1 & x_3 y_3 & x_2 y_3 & x_1 y_3 & x_0 y_3 \\
\end{array}
\]

Diagram showing the 2’s complement multiplication process with FA and HA blocks.
Carry-Save Addition

• Speeding up multiplication is a matter of speeding up the summing of the partial products.

• “Carry-save” addition can help.

• Carry-save addition passes (saves) the carries to the output, rather than propagating them.

• Example: sum three numbers, \(3_{10} = 0011, 2_{10} = 0010, 3_{10} = 0011\)

\[
\begin{align*}
3_{10} & \quad 0011 \\
+ 2_{10} & \quad 0010 \\
\text{c} \quad 0100 & = 4_{10} \\
\text{s} \quad 0001 & = 1_{10}
\end{align*}
\]

\[
\begin{align*}
3_{10} & \quad 0011 \\
\text{c} \quad 0010 & = 2_{10} \\
\text{s} \quad 0110 & = 6_{10} \\
1000 & = 8_{10}
\end{align*}
\]

• In general, carry-save addition takes in 3 numbers and produces 2.

• Sometimes called a “3:2 compressor”: 3 input signals into 2 in a potentially lossy operation

• Whereas, carry-propagate takes 2 and produces 1.

• With this technique, we can avoid carry propagation until final addition
Bit-serial Multiplier

- Bit-serial multiplier (n² cycles, one bit of result per n cycles):

- Control Algorithm:

```plaintext
repeat n cycles {  // outer (i) loop
    repeat n cycles{  // inner (j) loop
        shiftA, selectSum, shiftHI
    }
    shiftB, shiftHI, shiftLOW, reset
}
```

*Note:* The occurrence of a control signal x means x=1. The absence of x means x=0.
Booth recoding

(On-the-fly canonical signed digit encoding!)

<table>
<thead>
<tr>
<th>$B_{K+1}$</th>
<th>$B_K$</th>
<th>$B_{K-1}$</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>add 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>add $A$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>add $A$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>add $2A$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sub $2A$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>sub $A$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sub $A$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>add 0</td>
</tr>
</tbody>
</table>

$B_{K+1,K}A = 0A \rightarrow 0$
$= 1A \rightarrow A$
$= 2A \rightarrow 4A - 2A$
$= 3A \rightarrow 4A - A$

A “1” in this bit means the previous stage needed to add $4A$. Since this stage is shifted by 2 bits with respect to the previous stage, adding $4A$ in the previous stage is like adding $A$ in this stage!
Canonic Signed Digit Representation

- CSD represents numbers using 1, \(\bar{1}\), & 0 with the least possible number of non-zero digits.
  - Strings of 2 or more non-zero digits are replaced.
  - Leads to a unique representation.

- To form CSD representation might take 2 passes:
  - First pass: replace all occurrences of 2 or more 1’s:
    - 01..10 by 10..\(\bar{1}\)0
  - Second pass: same as above, plus replace 01\(\bar{1}\)0 by 0010 and 01\(\bar{1}\)10 by 00\(\bar{1}\)0

- Examples:
  - 0010111 = 23
  - 0011001 = 32 - 8 - 1
  - 011101 = 29
  - 1011010 = 64 - 8 - 2
  - 0101001 = 32 - 8 - 1
  - 00\(\bar{1}\)0\(\bar{1}\)0 = 0101001 = 32 - 8 - 1
  - 100\(\bar{1}\)0\(\bar{1}\) = 32 - 4 + 1
  - 100\(\bar{1}\)0\(\bar{1}\)0 = 64 - 8 - 2

- Can we further simplify the multiplier circuits?
Log Shifter / Rotator

- Log(N) stages, each shifts (or not) by a power of 2 places, \( S = [s_2; s_1; s_0] \):
Barrel Shifter

Cost/delay?
- (don’t forget the decoder)
Controller using Counters

- **State Transition Diagram:**
  - Assume presence of two binary counters. An “i” counter for the outer loop and “j” counter for inner loop.

  TC is asserted when the counter reaches its maximum count value. CE is “count enable”. The counter increments its value on the rising edge of the clock if CE is asserted.
Synchronous Counters

- How do we extend to n-bits?
- Extrapolate $c^+ = d^+ \oplus abc$, $e^+ = e \oplus abcd$

- Has difficulty scaling (AND gate inputs grow with n)

- CE is “count enable”, allows external control of counting,
- TC is “terminal count”, is asserted on highest value, allows cascading, external sensing of occurrence of max value.
Ring Counters

- “one-hot” counters
  0001, 0010, 0100, 1000, 0001, ...

“Self-starting” version:
Clock Constraints in Edge-Triggered Systems

If launching edge is late and receiving edge is early, the data will not be too late if:

\[ t_{\text{clk-q, max}} + t_{\text{logic, max}} + t_{\text{setup}} < T_{\text{CLK}} - t_{JS,1} - t_{JS,2} + \delta \]

Minimum cycle time is determined by the maximum delays through the logic

\[ t_{\text{clk-q, max}} + t_{\text{logic, max}} + t_{\text{setup}} - \delta + 2t_{JS} < T_{\text{CLK}} \]

Skew can be either positive or negative

Jitter \( t_{JS} \) usually expressed as peak-to-peak or \( n \times \) RMS value
Clock Constraints in Edge-Triggered Systems

If launching edge is early and receiving edge is late:

\[ t_{clk-q,min} + t_{logic,min} - t_{JS,1} > t_{hold} + t_{JS,2} + \delta \]

Minimum logic delay

\[ t_{clk-q,min} + t_{logic,min} > t_{hold} + 2t_{JS} + \delta \]

(This assumes jitter at launching and receiving clocks are independent – which usually is not true)
Sources of clock uncertainty
H-Tree

Equal wire length/number of buffers to get to every location
No voltage source is ideal - $||Z|| > 0$

- Two principal elements increase $Z$:
  - Resistance of supply lines (IR drop)
  - Inductance of supply lines ($L \cdot \frac{di}{dt}$ drop)
Types of Faults in Digital Designs

• **Design Bugs** (function, timing, power draw)
  – detected and corrected at design time through testing and verification (simulation, static checks)

• **Manufacturing Defects** (violation of design rules, impurities in processing, statistical variations)
  – post production testing for sorting
  – spare on-chip resources for repair

• **Runtime Failures** (physical effects and environmental conditions)
  – assuming design is correct and no manufacturing defects
Hamming Error Correcting Code

• Use more parity bits to pinpoint bit(s) in error, so they can be corrected.

• Example: Single error correction (SEC) on 4-bit data
  – use 3 parity bits, with 4-data bits results in 7-bit code word
  – 3 parity bits sufficient to identify any one of 7 code word bits
  – overlap the assignment of parity bits so that a single error in the 7-bit word can be corrected

• Procedure: group parity bits so they correspond to subsets of the 7 bits:
  – \( p_1 \) protects bits 1,3,5,7
  – \( p_2 \) protects bits 2,3,6,7
  – \( p_3 \) protects bits 4,5,6,7

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
p_1 & p_2 & d_1 & p_3 & d_2 & d_3 & d_4 \\
\end{array}
\]

Bit position number

\[
\begin{align*}
001 &= 1_{10} \\
011 &= 3_{10} \\
101 &= 5_{10} \\
111 &= 7_{10} \\
010 &= 2_{10} \\
011 &= 3_{10} \\
110 &= 6_{10} \\
111 &= 7_{10} \\
100 &= 4_{10} \\
101 &= 5_{10} \\
110 &= 6_{10} \\
111 &= 7_{10} \\
\end{align*}
\]

Note: number bits from left to right.
The End.

- Special thanks to our GSIs: Quincy and Tan.
- Good luck on the final.
- Thanks for a great semester!