Lecture 17: State
State...

- For digital design, we are always building finite state machines
  - Just the state can be damn large when you include external memory
  - So how do we build state?
- 4 basic state elements:
  - Registers
    - Fast, large, little array overhead
  - SRAM
    - Fairly fast, smaller, significant overhead
  - DRAM
    - Fairly slow, very dense, significant overhead
  - FLASH/EEPROM
    - Glacial, very dense, static, significant overhead
State Requires Storage & Access...

- Either as a feedback loop
  - Latches, Registers, SRAM
  - "Static"

- Or as a stored property
  - Electrons on a capacitor for DRAM
  - Electrons injected into a floating gate for FLASH

- Feedback loop...
  - Can just read the output directly (but there are tricks to speed it up)

- Stored property...
  - Need measurement circuits
Basic Static Storage Element

- If $D$ is high, $D_b$ will be driven low
  - Which makes $D$ stay high
- Positive feedback
**Positive Feedback: Bi-Stability**

\[ V_{o1} = V_{i2} \]

\[ V_{o2} = V_{i1} \]

(a)

(b)
Meta-Stability

Gain should be larger than 1 in the transition region:
For flip-flops we hate metastability...
But for DRAM we love it (as we will see soon)!
Reminder: Latch versus Register (Flip-flop)

◆ Latch: level-sensitive
  clock is low - hold mode
  clock is high - transparent

◆ Register: edge-triggered
  stores data when clock rises

Latch: level-sensitive

- Clock is low - hold mode
- Clock is high - transparent

Register: edge-triggered

- Stores data when clock rises
Creating a Latch & Flip-Flop...

- Take our basic static storage element...
  - And make it switchable between storage
- When clock is low, have the feedback loop active
- When lock is high, Q gets D
- Can then chain 2 latches to make a Flip/Flop
  - One negative clocked, one positive
Adding an Enable

- A very common motif
  ```verilog
  always @posedge clk begin
    q := en ? {something} : q
  end
  ```
  Namely, only update Q if an additional enable signal is high.

- Just simply add a higher level feedback loop with a mux.
  ```verilog
  ```
  {see board}
Creating a Register File... Reading

- Easiest thing is to start with a flip-flop...
- For reading, just have a MUX on the output to select the register in question
  - A common motif for all array memories: If you want the $n^{th}$ column/entry, you need a mux to select which one
- If done as flip/flops, this is simple...
Creating A Regfile... Write Enable

• We need to decode/demux the register address to determine which register to write...
  • Decoder is a very common operation:
    Take an \( n \) bit number and assert a different line for each bit
    Basically each output is an AND gate

• Commonly done with a "predecoder"
  • Compute \( \overline{A_0A_1}, \overline{A_0A_1}, \overline{A_0A_1}, A_0A_1 \)
  • Compute \( A_2A_3, A_2A_3, A_2A_3, A_2A_3 \)
  • Then do the AND of the intermediaries
Predecoder and Decoder
But Flip-Flops are big...

• 12 transistors for both latches...
  • Plus 2 to create a negated clock (can be shared)
• Plus the multiplexor to add an enable...
  • Since we don't want to always write
• Can we use the basic static element...
  • And make something more compact?
• Yes: SRAM
SRAM Memory Cell

Complementary data values are written (read) from two sides.
6-Transistor CMOS SRAM Cell
Compact cell
Bitlines: M2
Wordline: bootstrapped in M3
Initially precharge both the bit lines to VDD
BL gets pulled down if storing a 1, ~BL if its a 0
There is a fair amount of capacitance, must not overwhelm the inverter...
CMOS SRAM Analysis (Read)

\[ V_{DD} \]

\[ Q = 0 \]

\[ Q = 1 \]

\[ M_1 \]

\[ M_5 \]

\[ M_4 \]

\[ M_6 \]

\[ BL \]

\[ WL \]

\[ V_{DD} \]

\[ C_{bit} \]
CMOS SRAM Analysis (Write is a fight)

\[ \overline{BL} = 1 \quad Q = 0 \quad \overline{BL} = 0 \quad Q = 1 \]

\[ V_{DD} \quad W_L \]

\[ M_5 \quad M_4 \quad M_6 \quad M_1 \]
SRAM Column
Periphery

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

- Expands K address lines into N word lines
- A perfect example of logic/wire optimization
- Typically implemented in hierarchical fashion
- Area/Energy Trade-off
Speeding the Read: Sense-Amp

• It takes a fair bit of time to pull down the appropriate bitline...
  • But really, why not just detect a difference

• Idea: Precharge a reference and then look for a small change
Differential Sense Amplifier

Directly applicable to SRAMs
Differential Sensing — SRAM

(a) SRAM sensing scheme

(b) Two stage differential amplifier
Column Decoder

- Basically a multiplexer
SRAM Read Cycle

- **Precharge** all the bitlines to Vdd...
- Then assert the word line from the decoder
  - One or the other bit lines starts to drop towards ground...
  - Sense-amp uses this to amplify the result quickly
SRAM Write Cycle

• Actively drive the bitlines to the correct value
  • Then assert the correct word line
• Result is that it will flip the data to the correct value
Drive data bus **only when** clock is low

- Ensures address are stable for writes
- Prevents bus contention
- Minimum clock period is twice memory access time
Adding More Ports

- Just add a separate set of row selection decoders, bitlines, wordlines, sense-amps...
- But keep the common cell
DRAM...

- Damn, that SRAM cell is still big...
  - 6 transistors!!!!
- But we can also build small capacitors...
  - And small capacitors can hold their charge...
- IDEA: Let's store using *dynamic* logic...
  - Memory as state of change on a capacitor
- Sense a very small amount of charge...
  - And when we check it, we can then update it
1-Transistor DRAM Cell

Write: $C_S$ is charged or discharged by asserting WL and BL.
Read: Charge redistribution takes place between bit line and storage capacitance.

Voltage swing is small; typically around 250 mV.

$$\Delta V_1 = (V_{BIT1} - V_{PRE}) \frac{C_S}{C_S + C_{BL}} = \left(\frac{V_{DD}}{2} - V_T\right) \frac{C_S}{C_S + C_{BL}}$$

$$\Delta V_0 = (V_{BIT0} - V_{PRE}) \frac{C_S}{C_S + C_{BL}} = -\frac{V_{DD}}{2} \frac{C_S}{C_S + C_{BL}}$$

$C_S \ll C_{BL}$
**Latch-Based Sense Amplifier (DRAM)**

- Initialized in its meta-stable point with EQ
  - We also precharge the bit lines to the metastable point
- Once adequate voltage gap created, sense amp enabled with SE
- Positive feedback quickly forces output to a stable operating point.
  - Also acts to "write back" the value by driving the bit lines apart
What about that other bitline?

- We want something of the same capacitance as the bitline
  - Idea: Hey, we have a reference: The next column
- So we only read 1/2 the cells on a row for a given read
  - And use the other set of bit lines for reference
What About "Leakage"

- Those capacitors aren't great...
  - They will slowly lose charge over time...

- Idea: Just read *every cell* at a regular interval
  - This is called DRAM refresh

- Highly temperature dependent:
  - Hot DRAM tends to flip
What About Errors?

- Redundancy & sparing for manufacturing faults...
  - Check all the cells. If any fail in a column, replace that column with a spare column
- ECC memory for transient faults
  - EG, radioactive decay, cosmic ray, etc...
  - Cheap systems don't use ECC, but you should
Advanced 1T DRAM Cells

- Trench Cell
- Stacked-capacitor Cell

- Cell Plate Si
- Capacitor Insulator
- Refilling Poly
- Storage Node Poly
- Si Substrate
- 2nd Field Oxide
- Word line
- Insulating Layer
- Cell plate
- Capacitor dielectric layer
- Transfer gate
- Isolation
- Storage electrode
A “bank” of 128 Mb (512Mb chip -> 4 banks)

In reality, 16384 columns are divided into 64 smaller arrays.

13-bit row address input

16384 columns

8192 rows

134 217 728 usable bits (tester found good bits in bigger array)

16384 bits delivered by sense amps

Select requested bits, send off the chip
So basic DRAM read operation...

- Precharge all the bit lines in a block to $0.5V_{dd}$
- Enable the word line for the desired row
  - "Row Access"
- Activate the sense-amps
  - Acts to read the bits and restore the bits
- Once the sense-amps have the row
  - Do the column access to get the sub pieces within the row
- A write will start with a read...
  - Then override the bits you want to set
“Sensing” is row read into sense amps

13-bit row address input

1 of 8192 decoder

8192 rows

134,217,728 usable bits (tester found good bits in bigger array)

16384 columns

16384 bits delivered by sense amps

Select requested bits, send off the chip

Slow! A 2.5ns period DRAM (400 MT/s) can do row reads at only 55 ns (18 MHz).

DRAM has high latency to first bit out. A fact of life.
Latency is not the same as bandwidth!

What if we want all of the 16384 bits?

In row access time (55 ns) we can do 22 transfers at 400 MT/s.

16-bit chip bus -> 22 x 16 = 352 bits << 16384

Now the row access time looks fast!

Thus, push to faster DRAM interfaces.

13-bit row address input

13-bit decoder

8192 rows

16384 columns

134,217,728 usable bits
(tester found good bits in bigger array)

16384 bits delivered by sense amps

Select requested bits, send off the chip.
What if we want all of the 16384 bits?
The “we” for a CPU would be the program running on the CPU.
Recall Amdalh’s law: If 20% of the memory accesses need a new row access ... not good.

13-bit row address input
doing 1 of 8192 decoder rows (tester found good bits in bigger array)

16384 columns
16384 bits delivered by sense amps
Select requested bits, send off the chip

Sadly, it’s rarely this good ...
DRAM latency/bandwidth chip features

Columns: Design the right interface for CPUs to request the subset of a column of data it wishes:

- 16384 bits delivered by sense amps
- Select requested bits, send off the chip

Interleaving: Design the right interface to the 4 memory banks on the chip, so several row requests run in parallel.

Bank 1  Bank 2  Bank 3  Bank 4
Off-chip interface for a Micron DDR part...

A clocked bus: 200 MHz clock, data transfers on both edges (DDR).

Note! This example is best-case! To access a new row, a slow ACTIVE command must run before the READ.

DRAM is controlled via commands (READ, WRITE, REFRESH, ...)

Synchronous data output.
Opening a row before reading ...

Auto-Precharge READ

CK#
CK

CKE

Command

NOP

ACT

NOP

READ

NOP

NOP

NOP

NOP

ACT

Address

RA

Col n

RA

A10

RA

RA

Bank address

Bank x

Bank x

Bank x

DQS, DQS#

DQ6

15 ns

15 ns

55 ns between row opens.
However, we can read columns quickly

Note: This is a “normal read” (not Auto-Precharge). Both READs are to the same bank, but different columns.
Why can we read columns quickly?

Column reads select from the 16384 bits here.

13-bit row address input

8192 rows

16384 columns

134 217 728 usable bits (tester found good bits in bigger array)

16384 bits delivered by sense amps

Select requested bits, send off the chip
Interleave: Access all 4 banks in parallel

Interleaving: Design the right interface to the 4 memory banks on the chip, so several row requests run in parallel.

Can also do other commands on banks concurrently.
The SDRAM interface evolution

• Initially: 1b transferred per data line per clock cycle
• DDR: Double Data Rate: 2b per clock
  • Rising & falling edge
• DDR2: 4b per clock
• DDR3: 8b per clock
• DDR4: 16b per clock!
• But the latency hasn't really improved much in a decade+!!!
• So if you are touching DRAM, try to make everything sequential
Only part of a bigger story ...
Only part of a bigger story ...

CS 250  L10: Memory
(A) Without access scheduling (56 DRAM Cycles)

(B) With access scheduling (19 DRAM Cycles)

DRAM Operations:

P: bank precharge (3 cycle occupancy)
A: row activation (3 cycle occupancy)
C: column access (1 cycle occupancy)

From: Memory Access Scheduling
And Rowhammer....

• These DRAM cells are not *perfectly* isolated...
  • In reading a row, there is a chance that it could flip a bit in a different row

• Rowhammer is a hardware attack
  • Repeatedly read the same DRAM row to cause bit-flips elsewhere
  • Asking the OS to effectively fill the memory with page tables...
    • And then when a bit flips, it will cause the page table to be mapped into the process address space...
    • And once you do that, you win!
The physics of FLASH memory

Two gates. But the middle one is not connected.

1. Electrons “placed” on floating gate stay there for many years (ideally).
2. 10,000 electrons on floating gate shift transistor threshold by 2V.
3. In a memory array, shifted transistors hold “0”, unshifted hold “1”. 
A high drain voltage injects "hot electrons" onto floating gate.

A high gate voltage "tunnels" electrons off of floating gate.

1. Hot electron injection and tunneling produce tiny currents, thus writes are slow.

2. High voltages damage the floating gate. Too many writes and a bit goes "bad".
NAND Flash Memory
Flash: Disk Replacement

Presents memory to the CPU as a set of pages.

Page format:

2048 Bytes + 64 Bytes

(user data) (meta data)

1GB Flash: 512K pages
2GB Flash: 1M pages
4GB Flash: 2M pages

Chip “remembers” for 10 years.
Reading a Page ...

33 MB/s Read Bandwidth

Read Operation

Bus Control

Flash Memory

8-bit data or address
(bi-directional)

Samsung
K9WAG08U1A

Clock out page bytes:
52,800 ns

Page address in: 175 ns

First byte out: 10,000 ns

Flash Memory

CLE

CE

WE

ALE

RE

I/Ox

R/B

00h Column Add1 Column Add2 Row Add1 Row Add2 Row Add3 30h

Column Address Row Address

Busy

Dout N Dout N+1 Dout M

Page address in: 175 ns

First byte out: 10,000 ns

Clock out page bytes: 52,800 ns

Flash Memory

CLE

CE

WE

ALE

RE

I/Ox

R/B

00h Column Add1 Column Add2 Row Add1 Row Add2 Row Add3 30h

Column Address Row Address

Busy

Dout N Dout N+1 Dout M

Page address in: 175 ns

First byte out: 10,000 ns

Clock out page bytes: 52,800 ns

Flash Memory

CLE

CE

WE

ALE

RE

I/Ox

R/B

00h Column Add1 Column Add2 Row Add1 Row Add2 Row Add3 30h

Column Address Row Address

Busy

Dout N Dout N+1 Dout M

Page address in: 175 ns

First byte out: 10,000 ns
Where Time Goes

Figure 1. K9K8G08U0A Functional Block Diagram

Page address in: 175 ns

Page address in: 175 ns

First byte out: 10,000 ns

Clock out page bytes: 52, 800 ns
Writing a Page ...

A page lives in a **block** of 64 pages:
- 1GB Flash: 8K blocks
- 2GB Flash: 16K blocks
- 4GB Flash: 32K blocks

To write a page:

1. **Erase all pages in the block** (cannot erase just one page).  
   **Time:** 1,500,000 ns

2. **May program each page individually, exactly once.**  
   **Time:** 200,000 ns per page.

**Block lifetime:** 100,000 erase/program cycles.
Block Failure

Even when new, not all blocks work!

1GB: 8K blocks, 160 may be bad.
2GB: 16K blocks, 220 may be bad.
4GB: 32K blocks, 640 may be bad.

During factory testing, Samsung writes good/bad info for each block in the meta data bytes.

<table>
<thead>
<tr>
<th>Block 0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 0</td>
<td></td>
</tr>
<tr>
<td>Page 1</td>
<td></td>
</tr>
<tr>
<td>Page 62</td>
<td></td>
</tr>
<tr>
<td>Page 63</td>
<td></td>
</tr>
</tbody>
</table>

2048 Bytes + 64 Bytes

(user data) + (meta data)

After an erase/program, chip can say "write failed", and block is now "bad". OS must recover (migrate bad block data to a new block). Bits can also go bad "silently" (!!!).
Flash controllers: Chips or Verilog IP ...

Flash memory controller manages write lifetime management, block failures, silent bit errors ...

Software sees a “perfect” disk-like storage device.
Actually Using Memory

- Two options:
  - Directly instantiate memory blocks:
    Can either use IP generators or instantiate primitives directly
  - Write Verilog with something the tools can infer

```verilog
parameter n = 4;
parameter w = 8;
reg [w-1:0] reg_array [2**n-1:0];
always @ posed clk begin
  if (we) reg[write_addr] <= din;
end
always @* begin
  dout <= reg[read_addr]
end
```

- should be inferred as a simple dual-port memory:
  One synchronous write port, one asynchronous read port
Some Types of Memory...

• Single port:
  • One address port, one data in port, one data out port
  • Can read or write

• Simple dual port:
  • Two address ports, one for reading, one for writing
    • Very good for implementing FIFOs!

• True dual port:
  • Two address ports, both can be used for reading or writing

• Suggestion for Xilinx version of the project:
  • Best way to do the processor reg-file is instantiate simple-dual-port memories