32k x 8 SRAM: 15 bit address
- use top 9 bits to index into 8 smaller SRAMs, producing 8 64-bit values
- use bottom 6 bits to multiplex further, choosing 1 bit of 64-bit value

address [14:63] → 9:4 size decoder → 8:4 size decoder

1 bit per SRAM/mux

DRAMs
- "dynamic" - has to be refreshed periodically - written back
- value is stored as a charge in a capacitor (transistor)
- denser, cheaper
- refresh happens automatically by memory controller
- two-level decoding structure refreshes whole row at a time
- consumes 1% – 2% of available DRAM cycles (1997)
DRAMS are organized into rows x columns
-

- select pins by sending Row Address x Column Address on same input lines, one after the other.
- enabling lines: RAS - row access strobe
  CAS - column access strobe

4 M x 1

4-Mbit DRAM

22 bits

Individual DRAM Cell

Write:
Assert word line
(bit line to capacitor)
* bit line = charged or discharged

Read:
Assert word line & bit
Mid-level voltage on
bit line. If voltage is
(a little) different from bit line,
charge was 1, no voltage increase
charge was 0.
Machine Addition/Subtraction
  - Subtraction is generally addition w/ negation
  - Need to be careful of overflow from addition
  2 Cases:
  1. Two large positives → negative
  2. Large negative → positive
  - Hardware should check these cases.
  - Sometimes generate traps.
  - Depends on machine, OS, & language.

Machine logical operations
  bitwise and, bitwise or, xor, etc.

Constructing an ALU
  - From and, or, inverting mux

- 1 bit logical and/or

```
\[ \text{select operation} \]
\[ a \] \quad \text{and/or} \quad \text{result} \\
\[ b \]
```
What about addition?

1-bit adder: 3 inputs, 2 outputs
\[ a, \bar{b}, \text{carry in} \quad \text{sum, carry out} \]

Express as nested boolean functions:
\[
\text{sum} = (a \cdot \bar{b} \cdot \text{cin}) + (\bar{a} \cdot b \cdot \text{cin}) + (a \cdot \bar{b} \cdot \text{cin}) + (a \cdot b \cdot \text{cin})
\]
\[
\text{Cout} = (a \cdot b) + (a \cdot \text{cin}) + (b \cdot \text{cin})
\]

Boolean circuits follow from these equations:

- Example computation of carry-out:

  ![Diagram of carry-out computation](image)

Represent as:

![Basic circuit symbol](image)
An ALU

1-bit - does AND, OR, +

32-bit - need to cascade the carry-out

Cripple carry
Subtraction: the beauty of 2's complement

\[ a - b = a + (b + 1) = (a + b) + 1 \]

- so negate bits of \( b \), set \( C_{in} \) to 1.
- need negation logic

1-bit ALU becomes:

\[ \text{Result} = \begin{cases} \text{Binop} & \text{if } a < b \\ \text{Input} & \text{otherwise} \end{cases} \]

Together, Binum + 2-bit op becomes a 3-bit op select

What about comparison?

\[ \text{MIPS: } \text{slt } \$1, \$2, \$3 \quad \text{SLT } \begin{cases} 1 & \text{if } \$1 < \$2 < \$3 \\ 0 & \text{otherwise} \end{cases} \]

use subtraction:

\[ a < b \iff a - b < 0 \]

- direct most significant bit of result of subtraction to the least significant bit of the output
- send 0 to all other output bits
Building a datapath
- i.e. mechanism for loading, storing, running instructions & data
- involves registers, memories, ALU, etc.

Portion for Fetching instructions:
- PC holds address of instruction
- After fetch, increment PC by 4

Datapath for "R-Type Instructions"
- 2 register operands (source)
- 1 register (destination)
Datepath for load/store instructions of the form:

- **lw**: `$reg1, offset($reg2)`, 16 bits
- **sw**: `$reg1, offset($reg2)`, 16 bits

Address is `$reg1` contents + offset, which can be negative!
Datapath for a conditional branch instruction: 

```
PC-relative

beq $reg1, $reg2, address
```