Computer Architecture

Organizational Stuff:
- Textbook
- Syllabus
- Web Page
- Mailing List
- Office Hours
- Assignments
- Mid-Term & Final Exam

"Computer Architecture" - definition
- ISA
- Microarchitecture
- the architecture as an abstraction

A little history - (Mechanical)

- Jacquard Loom - late 1700's
- Babbage's Difference & Analytical Engines - mid 1800's
- Hollerith's Census Counter 1890
- company eventually became IBM

- Electronic Digital
- ENIAC - mid 1940's (durin)
- Eckert-Mauchly
- Computing Ballistics Tables

EDSAC - stored program
- computer 1948
- stored program was an idea floating around;
- written up by J. Von Neumann
Improvements over the years:

**Moore's Law:** Density of circuits (gates) doubles every 18 months. Has held true since 1962 or so. Performance roughly follows.

"DRAM Growth Rule" - Capacity roughly quadruples every 3 years.

<table>
<thead>
<tr>
<th>Model</th>
<th>Year</th>
<th>Capacity</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Univac 11</td>
<td>1951</td>
<td>1900 addr/s</td>
<td>$5M</td>
</tr>
<tr>
<td>IBM 7090</td>
<td>1964</td>
<td>500,000</td>
<td>$4M</td>
</tr>
<tr>
<td>PDP-8</td>
<td>1965</td>
<td>330,000</td>
<td>$66K</td>
</tr>
<tr>
<td>IBM 360/44</td>
<td>1981</td>
<td>240,000</td>
<td>$1M</td>
</tr>
<tr>
<td>Intel Pentium 4</td>
<td>2002</td>
<td>2.5 GHz</td>
<td>$1K</td>
</tr>
</tbody>
</table>

Some terms: RISC, CISC, Pipeline, Vector, VLIW, ILP, Uniprocessor, Multiprocessor, Multicomputer.
Overall Organization of a Computer:

- Memory/Cache
- Processor/Registers
- Memory bus
- I/O devices
- I/O bus

Recognizing that voltage levels can represent binary values:
1. 0 vs 1
2. Transistors: - Semiconductors - Sometimes conduct

CMOS - 3 terminals: Source, Drain, Gate

- Two types of transistors
- Voltage is applied at the source
- N-type:
  - When voltage is high at the gate, the drain current is high (thus)
  - When voltage is low at the gate, drain current is low

- P-type:
  - Opposite behavior of N-type

So, can build an inverter:  
\[ \text{Vin} \rightarrow \text{Vout} \]
An inverter is just one example of a gate. Others are:

\[ \text{AND} \quad \text{OR} \quad \text{Not} \quad \text{NOR} \quad \text{NAND} \]

These correspond to boolean functions, which can be represented by truth tables: (1 = true, 0 = false)

<table>
<thead>
<tr>
<th>OR</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AND</th>
<th>0</th>
<th>1</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAND</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Truth tables can have arbitrarily many inputs. Multiple functions can be represented by a single table if multiple output columns.

<table>
<thead>
<tr>
<th>inputs</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>outputs</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: number of rows = \(2^\text{# of inputs}\)
Boolean functions can also be represented by formulas, where + is OR, ⋅ is AND, and ̅A is not A

In previous chart

\[ D = (A + B) \cdot ̅C \]
\[ E = B + C \]

These formulas form Boolean algebra, with the following laws:

**Identity**
\[ A + 0 = A \]
\[ A \cdot 1 = A \]

**Inversion**
\[ A \cdot ̅A = 0 \]
\[ A + ̅A = 1 \]

**Associativity**
\[ (A + B) + C = A + (B + C) \]
\[ (A \cdot B) \cdot C = A \cdot (B \cdot C) \]

**Commutativity**
\[ A \cdot B = B \cdot A \]
\[ A + B = B + A \]

**Distributivity**
\[ A \cdot (B + C) = (A \cdot B) + (A \cdot C) \]
\[ A + (B \cdot C) = (A + B) \cdot (A + C) \] (e. l. arithmetic)

**De Morgan's Laws**:
\[ (A + B) = ̅A \cdot ̅B \]
\[ (A \cdot B) = ̅A + ̅B \]
Now that we know about boolean functions, we can build circuits to represent them!

\[ A + (B \cdot C) \]

Any boolean function can be implemented using AND, OR, and NOT gates.

Also can be implemented using a single kind of gate, NAND.

\[ \overline{A} = \overline{A \cdot A} \]

\[ A \cdot B = (A \cdot B) \]

\[ A + B = (\overline{A \cdot B}) \]

The circuits that implement boolean functions are called combinational \underline{combinational logic}.

-no state result depends only on inputs

Address logical unit

Circuits that retain state are called sequential

-register \underline{logic}

-memory \underline{memory} -memory elements
Some useful combinational building blocks:

**Decoders**

- The \( N^{th} \) output line is asserted, where \( N \) is the value of the input.
- Implementing a 1-bit decoder:
  - \( \text{out}_0 = \overline{\text{IN}} \)
  - \( \text{out}_1 = \text{IN} \)

2-bit decoder:

\[
\begin{align*}
\text{out}_0 &= \overline{\text{INO}} \cdot \overline{\text{IN1}} \\
\text{out}_1 &= \overline{\text{INO}} \cdot \text{IN1} \\
\text{out}_2 &= \overline{\text{INO}} \cdot \text{IN1} \\
\text{out}_3 &= \text{INO} \cdot \text{IN1}
\end{align*}
\]

**Multiplexor (aka MUX) - a selector**

- Output gets the value of the \( \text{INO} \) input \( 2^{N-1} \) where \( N \) is the value of the selector.
Implementing a 2 input MAX

\[ \text{Out} = (IN_0 \cdot \overline{S}) + (IN_1 \cdot S) \]

4 input MAX:

\[ \text{Out} = (IN_0 \cdot \overline{S_0} \cdot S_1) + (IN_1 \cdot S_0 \cdot \overline{S_1}) + (IN_2 \cdot \overline{S_0} \cdot S_1) + (IN_3 \cdot S_0 \cdot S_1) \]

Notice the common case of a "sum of products," where the only negation is on a variable or at the output.

- "two level logic"
- easily represented as a Programmable Logic Array (PLA)
PLA - can represent many functions

Example: One-bit addition with carry in and carry out

<table>
<thead>
<tr>
<th>C_in</th>
<th>A</th>
<th>B</th>
<th>C_out</th>
<th>Rезульт</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Using ROMs for combinational circuits:
- Just store the truth table, where the n inputs represents the address and the m outputs represents the value stored.
- Less "efficient" than PLAs, since every combination of inputs is represented in the memory.

Arrays of logic

Generally, operations are performed on many bits — i.e., some operation on each bit.

![Diagram](Diagram)

Actually an array of 1-bit muxes.

Selecting among 2^n 32-bit inputs.
**Clocks**

- Need to indicate a change of state in sequential logic (i.e., storage logic)

- Regular change in voltage on a line

- Many circuits are edge triggered
  - Either rising or falling
  - Don't have to be, but easiest to reason about

```
State element 1 ➔ Combinational logic ➔ State element 2
```

- State element 1 must be stable here
- State edge 2 must be stable here

**Edge triggering allows this to occur in a single cycle.**
Memory Elements

Basic elements are flip-flops & latches

Unclocked latch: simplest
\[ S = R \text{ is set/reset} \]

- \[ R = 1, S = 0 \rightarrow Q = \overline{A}, \overline{Q} = A \]
- \[ S = I, R = 0 \rightarrow Q = 1, \overline{Q} = 0 \]
- \[ S = 0, R = 0 \rightarrow Q \text{ unchanged}, \overline{Q} \text{ unchanged} \]
- \[ S = I, R = 1 \rightarrow \text{undefined} \]

Clocked latches & flip-flops

- Clocked latch changes whenever inputs change & clock is asserted high.
- Clocked flip-flop changes only upon a clock edge.

Text only uses clocked flip-flops

D latch:

- When \( C \) is asserted, \( Q \) gets the value of \( D \).
- When \( C \) is asserted, \( Q \) retains the value of \( D \).
D Flip-Flop w/ falling edge trigger

D must be asserted long enough so that correct value of Q is propagated from the first to the second latch.

REGISTER FILES

register file - set of identical registers that can be written to or read from.

E.g. Two read, one write register file:
**REGISTER FILE IMPLEMENTATION**

SRAM: fast memory for caches

- **Height**: # of addressable locations
- **Width**: # of bits per location

**Example**:
- 256K x 1: 256K on 8-bit locations
- 32K x 8: 32K, 8-bit locations

Modern SRAMs are "narrow":
- 8 input
- 8 output
- 31 lines
We can't build SRAMs like we build register files.
- MUX is way too big (remember: actually an array of DEMUXs)
- Need to share output line among the memory elements, but can't have several writing at once.
- Need a 3-state buffer

\[
\text{in} \rightarrow \text{out} \quad \text{if enable} = 1, \text{then out} = \text{in} \\
\text{otherwise nothing is on the line ("high impedance state")}
\]

Selecting output:

Select 0 \quad \text{Out} \quad \text{enable}
\text{Select 1 \quad Data 1 \quad in} \quad \text{out}
\text{Select 2 \quad Data 2 \quad in} \quad \text{out}
\text{Select 3 \quad Data 3 \quad in} \quad \text{out}

out

Easily build a 4-bit latch using 3-state buffers:

\[ D_{\text{in}} \quad D_{\text{in}} \quad \text{enable} \rightarrow Q \]

\text{If enable} = 0, \text{nothing is on the line.}
Building SRAM

Idea: “row” selects address
“col” is a bit at the location

Implementation 4x2 SRAM

Larger SRAMs are built out of arrays of smaller SRAMs
32K x 8 SRAM: 15 bit address
- use top 9 bits to index into 8 smaller SRAMs, producing 8 44-bit values
- use bottom 6 bits to multiplex
  further, choosing 1 bit per 4-bit value

1 bit per SRAM/mux

DRAMs
- dynamic - has to be refreshed periodically - written back
- value is stored as a charge in a capacitor (transistor)
  - smaller, cheaper
- refresh happens automatically by memory controller
- two-level decoding structure refreshes whole page at a time
- consumes 1% - 2% of variable DRAM cycles (1997)
**DRAM** are organized into rows × columns.
- Same pins by sending row addresses × columns, selecting on same input lines, one after the other.
- Enabling lines: RAS - row access strobe
  CAS - column access strobe

4M x 1 dram (22 bits)

<table>
<thead>
<tr>
<th>Address</th>
<th>11-bit at a time</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>11-20 tapping</th>
<th>10 tapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input to Row decoder</td>
<td>Input to 2M x 208 array</td>
</tr>
</tbody>
</table>

Refresh

<table>
<thead>
<tr>
<th>Column Latch</th>
<th>2048x2</th>
</tr>
</thead>
</table>

1-bit unit

---

**Individual DRAM Cell**

| Word line | Write: Assert word line * (closing switch connecting bit line to capacitor)
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>capacitor</td>
<td>* Bit line: charged or discharged</td>
</tr>
</tbody>
</table>

| Bit line | Read: Assert word line & put
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mid-level voltage on bit line. If voltage is G (offt) (opposite from bit line), charge was 0, in voltage increases, charge was 1.</td>
</tr>
</tbody>
</table>
Assembly Code

- Give intro to compilation, assembling, linking, loading
- also assembly, machine code, microcode.
- hardwired vs. microprogrammed.

MIPS machine code

32 32-bit registers, numbered 0-31

always: $zero = 0 always contains 0

$0 - $7 = $4 - $7 used for parameter passing

$0, $1 = $2 - $3 return values

$10 = $19 = $8 - $15, $24, $25 caller-saved temp. registers

$50 - $57 = $16 - $23 callee-saved reg. for local variables

$fp = $29 stack pointer

$fp = $30 frame pointer

$ra = $31 return address

$gp = $28 points to global area

other reserved
Mips Instructions: 32 bits

Several formats

R-type ("register") instruction formats

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bit</td>
<td>5 bit</td>
<td>5 bit</td>
<td>5 bit</td>
<td>5 bit</td>
<td>6 bit</td>
</tr>
</tbody>
</table>

- **op**: operation code, ALU op, or other.
- **rs**: first register
- **rt**: second register
- **rd**: destination register
- **shamt**: shift amount or shift operation
- **funct**: special function

General format of R-type assembly instruction:

Anthelitic: `op dest, op1, op2`

So:

- **add**: `add $s1, $s2, $t2`
- **sub**: `sub $t2, $t3, $t1`

Load/Stores can also be in R-format

- Just means REG used to specify

<table>
<thead>
<tr>
<th>Address</th>
<th>Address</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>rd + shamt</td>
<td>funct bits</td>
</tr>
</tbody>
</table>

16b $54, 80($t2) - can load bytes as well
1000+4($t4) = 50 $t7, 1000($t4) - 16b/32b must be word aligned.
Comparison & Branching

Offset uses (d+short formats)

Comparison

set $s1, $s2, $s3
$s1 gets $0 if $s2 $s3

Branch

beq $s1, 0, 100, 8

-compile -that's it!
- no bit fole, etc. - up to compiler
- always relative addressing for branch instructions

Note: assembly programmers use labels, assembler generates relative offsets.
J-type instructions

jump  j 14100
  → absolute address
    op: 6 bits
    address: 26 bits
    → bit in word
    not bytes

jump-registers

jr  $s1
  32 bit address contained in register
  → better be word aligned
  RS register field is used
  op = 0, funct = 8

jump-and-link (procedure call)

jal  2424
  op: 6 bits
  address: 26 bits
  → words, not bytes.
I-type instructions (Immediate)

- 3rd operand is contained in the instruction itself—the value, not the register.

Format: op rs rt address/value
6bits 5bits 5bits 16bits

- Note that offset in lw/sb is word immediate

Examples:
addi $51, $t1, 5202
- No subi; just make immediate value negative

sli $t4, $53, 50

Questions:
How do you use immediate values larger than 16 bits? 32-bit immediate
How do you specify an address to load?

Choices:
1. Have loader place the desired 32-bit value in global memory, then load it.
   - slow memory access, immediate
2. Load-shift-add: load high 16 bits, shift, three instructions, add low bits
3. lui $s1, 9567; load upper immediate
   addi $s1, $s1, 8910
Computer Arithmetic

Quick review of binary numbers:

- Addition, subtraction show carry in/out

Signed numbers:
- Sign and magnitude: two 0's, addition is complicated
- Two's complement:

    non-negative:
    
    \[
    \begin{align*}
    000...00 &= 0_{10} \\
    000...01 &= 1_{10} \\
    011...11 &= 2^{31} - 1
    \end{align*}
    \]

    negative:

    demonstrate: subtract 1 from 0.

    \[
    \begin{align*}
    11...111 &= -1_{10} \\
    11...110 &= -2_{10} \\
    11...101 &= -3_{10} \\
    \vdots \\
    100...00 &= -2^{31}
    \end{align*}
    \]

One way to view a 2's complement # : $$(\text{bit}_{31} \times 2^{31}) + \left( \text{bit}_{30} \ldots \text{bit}_0 \right)$$
Forming the negative of a number: 1) subtract from 0 equivalent  2) flip $0^t + 1^s$ and add 1.

MIPS (and other serious processors) have signed and unsigned instructions.
- signed instructions treat numbers w/ leading 1 as large positive instead of negative.

Signed instructions need to preserve signs:
- shift right: replicate leading bit ("sign bit")
- shift left: preserve sign bit, signal overflow
- loading 16 bit signed value into 32-bit register: replicate sign bit.

Representing binary numbers in other bases:

- base 10 - works space (need 4 bits anyway)
- base 16 - hexadecimal
  - 0-9, a, b, c, d, e, f
  - a = 1010
  - b = 1011
  - c = 1100
  - d = 1101
  - e = 1110
  - f = 1111
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. Two large negatives → positive

- Hardware should check these cases.
- Sometimes generate traps.
- Depends on machine, OS, or language.

Machine logical operations
- bit-wise and, bit-wise or, xor, etc.

Constructing an ALU
- from and/or, invert, mux

- 1 bit logical and/or

\[
\begin{array}{c}
\text{select: operation} \\
\hline
\text{a} \\
\hline
\text{b} \\
\text{result}
\end{array}
\]
What about addition?

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

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

\[
\text{Cout} = (a \cdot b) + (a \cdot \text{cin}) + (b \cdot \text{cin})
\]

Boolean Circuits follow from these equations
eg. computation of carry-out:

Represent as
An ALU

1-bit - does AND, OR, +

32-bit - need to cascade the carry-out (ripple carry)

32-bit ALU diagram with carry-in and carry-out connections.
Subtraction: the beauty of 2's complement

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

- so negate bits of \( b \), set \( C_0 \) to 1.
- need negation logic

1-bit ALU becomes:

```
+  +  +
|   |   |
a + b + 1
  +  +  +
```

Together, Binary + 2-bit \( op \) becomes a 3-bit \( op \) select

What about comparison?

MIPS: `slt $s1, $s2, $s3`\[ s1 < s3 \]

we subtract:

\[
\begin{align*}
q &< b \iff a - b < 0 \\
\text{direct most significant bit of result of subtraction to the least significant bit of the output} \\
\text{send 0 to all other output bits}
\end{align*}
\]
Add an extra input line, "less", that is wired to 0 for all but the last 1-bit ALU.

- Op now has a "less" choice.
- Additional output, "set", which is ignored except for the set output of ALU31.
- "set" is the result of the addition.

Equality test

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

So, need to test if all outputs of subtraction are 0.
Problem w/ ripple carry: slow, has to propagate across the 32 ALU's. Sequences the additions.

Don't need to have ripple carries, can define results and carry output directly.

\[
\begin{align*}
S_{in} &= (a \cdot b) + (a \cdot c_{in}) + (b \cdot c_{in}) \\
C_{in} &= C_{out} = (a \cdot b) + (a \cdot c_{in}) + (b \cdot c_{in})
\end{align*}
\]

An example:

\[
\begin{align*}
C_{in} &= C_{out} = (a_0 \cdot b_0) + (a_0 \cdot c_{in}) + (b_0 \cdot c_{in}) \\
C_{in2} &= C_{out} = (a_0 \cdot b_0) + (a_0 \cdot c_{in1}) + (b_0 \cdot c_{in1}) \\
&= (a_0 \cdot b_0) + ((a_0 + b_0) \cdot c_{in2}) \\
&= (a_0 \cdot b_0) + ((a_0 + b_0) \cdot ((a_0 + b_0) \cdot c_{in2})) \\
C_{in+1} &= (a_i \cdot b_i) + ((a_i + b_i) \cdot c_{in+1})
\end{align*}
\]

"generate" "propagate"
Notation: Use 'C_i' to mean 'C_{i-1}'

Use the notion of the generate term and the propagate term to build a carry lookahead adder (i.e., fastest way for the result of 7-bit addition).

Let \( g_i = \text{generate} (a_i \cdot b_i) \)
\[ p_i = (g_i + b_i) \]

4-bit adder

\[ C_1 = g_0 + (p_0 \cdot c_0) \]
\[ C_2 = g_1 + (p_1 \cdot (g_0 + (p_0 \cdot c_0))) = g_1 + (p_1 \cdot g_0) + (p_1 \cdot p_0 \cdot c_0) \]
\[ C_3 = g_2 + (p_2 \cdot g_1) + (p_2 \cdot p_1 \cdot g_0) + (p_2 \cdot p_1 \cdot p_0 \cdot g_0) \]
\[ C_4 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) + (p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0) \]

Now, let's treat each 4-bit adder as a single block. - Can ripple carries between 4-bit adders, or - Perform carry lookahead at the next level of abstraction

Super generate, super propagate terms.
For 8-bit propagates, for a 16-bit adder made of 4 4-bit adders:

\[
P_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0
\]

\[
P_1 = p_7 \cdot p_6 \cdot p_5 \cdot p_0
\]

\[
P_2 = p_{11} \cdot p_{10} \cdot p_9 \cdot p_0
\]

\[
P_3 = p_{15} \cdot p_{14} \cdot p_3 \cdot p_2
\]

where each \( p_i \) is as defined previously within a 4-bit adder.

The carry-in to a 4-bit adder only propagates to carry-out if all \( p_i \) inside adder are true.

Super-generates:

\[
G_0 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0)
\]

\[
G_1 = g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4)
\]

etc.

Now we can define the carry-ins to the 4-bit adders as:

\[
C_1 = G_0 + (P_0 \cdot C_0)
\]

\[
C_2 = G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot C_0)
\]

\[
C_3 = G_2 + (P_2 \cdot G_1) + (P_2 \cdot P_1 \cdot G_0) + (P_2 \cdot P_1 \cdot P_0 \cdot C_0)
\]

\[
C_4 = G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) + (P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0)
\]
Important points: each p, q, P, G do not require the carry-in bit.
- can be computed as soon as operands are available

- so, we can build the device that computes each \( C_i \)
  given each \( P_i \), \( G_i \), and \( C_0 \).

\[
\text{"carry lookahead unit"}
\]
Have we really speeded things up?

- 16-bit ripple carry: 1-bit adder takes two gate delay to compute $C_{out}$ from $9.5 + C_{in}$ (refer to picture)
- 16 1-bit adders are chained together, so a 32-gate delay

16-bit carry look-ahead, as just described:

- Each $P_i$ and $q_i$ take 1 gate (or an AND)
- Each $P_i$ is an AND with 4 inputs
- Best: 1 gate
- Worst: 2 gates

Each

- best: 2 gates (AND's then OR)
- worst: 5 gates: 4-input AND then last 2-input OR

Most complicated $C$ is $C_2$:
- Best: 2 gates (AND's then OR)
- Worst: 5 gates (show it)

Total:
- Best: 5 gate delay = most likely
- Worst: 10 gate delay
**Multiplication**

- Review of multiplication in decimals and binary

- Basic idea:

  ```c
  product = 0
  for(i = 0; i < 32; i++)
  
  if (multiplier & 1)
    product += multiplicand;

  multiplicand = multiplicand << 1;
  multiplier = multiplier >> 1;
  ```

Note: multiplicand x product must be stored in 64-bit registers, assuming 32-bit multiplication.

Assume registers have shift-left *shift right* lines, one write line.
2nd Version

Since only 32 bits of the multiplicand are relevant, keep it in a 32-bit register.
- add it to the upper 32 bits of the product
- then shift the product to the right

Makes sense:
- result of first addition ends up in lowest 32 bits
- result of last addition in top 32 bits

3rd Version

- Note that lowest 32 bits of the initial product value go unused (get shifted off)
- store the multiplicand in there!
  - you are checking the lowest bit of the product register, writing to the upper 32 bits each time

![Diagram]
Signed Multiplication

One way:
- Convert to positive numbers, remembering the signs
- Perform previous algorithm, but on 31-bit numbers
- Convert to proper sign.

Another way:
- Be sure to keep proper signs in the product register
- i.e., when you shift right, perform sign extension

A third way: Booth's Algorithm
- Reduces the number of adds required
- Preserves sign.

Basic Idea:

Note that
\[ \underbrace{111 \ldots 1}_{k} = 2^k - 1 \]

So \[ (\underbrace{111 \ldots 1}_{k} - 1) \times m = 2^k m - m \]
Booth's Algorithm, continued:

Thus, when the multiplier looks like \( \overline{011...1} \), subtract \( m \) from the multiplicand from the product. Then keep shifting the product & multiplier until the multiplier contains \( \overline{0} \) (i.e. you've shifted \( k \) times), then add \( m \) to the multiplicand. This gives \( \sum_{i=0}^{k-1} 2^i m = \overline{011...1} \times m \) to the multiplicand.

So, only need to add (or subtract) when the lowest bit of the multiplier changes value after a shift.
**DIVISION**

Review of decimal & binary division...

Terms: \[ \frac{\text{dividend}}{\text{divisor}} = \text{quotient}, \text{ with a remainder} \]

Actually: \[ \text{dividend} = (\text{quotient} \times \text{divisor}) + \text{remainder} \]

Like multiplication, three versions of division.

---

**Basic Idea**

64 bits → 32 bits

quotient = 0; divisor = Actual_divisor << 32;
remainder = dividend

for (i = 0; i < 32; i++) {
    if (remainder >= divisor) {
        quotient = (quotient << 1) | 1;
        remainder = remainder - divisor;
    } else {
        quotient = (quotient << 1);
    }
    divisor = divisor >> 1;
}
Version 1

Direct implementation of above algorithm.
- except the comparison is a subtraction
  and then, if necessary, a re-adding

1-e. the lines:

```c
if (remainder >= divisor) {
    quotient = (quotient << 1) / 1;
    remainder = remainder - divisor;
}
else {
    quotient = quotient << 1;
}
```

become:

```c
remainder = remainder - divisor;
if (remainder < 0) {
    remainder = remainder + divisor;
    quotient = quotient << 1;
}
else {
    quotient = (quotient << 1) / 1;
}
```