The Berkeley Remix Podcast, Season 4, Episode 2, “Berkeley Lightning: A Public University’s Role in the Rise of Silicon Valley”

IC chip from Hewlett Packard 34C Calculator, 1979-83. Some of the calculator’s revolutionary features were designed by UC Berkeley computer scientist William M. Kahan
Review

• Finite state machines: Common example of sequential logic
  • Moore’s machine: Output depends only on the current state
  • Mealy’s machine: Output depends on the current state and the input

• Large state machines can be factored

• Common Verilog patterns for FSMs

• Common job interview questions 😊
Combinational Logic (Take 3)
Canonical Forms

- Two types:
  - Sum of Products (SOP)
  - Product of Sums (POS)

- Sum of Products
  - a.k.a Disjunctive normal form, minterm expansion
  - Minterm: a product (AND) involving all inputs
  - SOP: Summing (ORing) minterms for which the output is True

<table>
<thead>
<tr>
<th>Minterms</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
<th>f'</th>
</tr>
</thead>
<tbody>
<tr>
<td>a'b'c'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a'b'c'</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a'b c'</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a'b c</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a b'c'</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a b'c</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a b c'</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a b c</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

One product (and) term for each 1 in f:
\[
f = a'b'c + ab'c' + ab'c + abc' + abc
\]
\[
f' = a'b'c' + a'b'c + a'bc'
\]
Sum of Products (cont.)

• Canonical Forms are usually not minimal:
  • Example:

\[
\begin{align*}
f &= a'bc + ab'c' + ab'c + abc' + abc \\
    &= a'bc + ab' + ab \\
    &= a'bc + a \\
    &= a + bc \\
\end{align*}
\]

\[
\begin{align*}
f' &= a'b'c' + a'b'c + a'bc' \\
    &= a'b' + a'bc' \\
    &= a' (b' + bc') \\
    &= a' (b' + c') \\
    &= a'b' + a'c'
\end{align*}
\]

\[a + a'b = a + b\]

• Recall distributive theorem
  \[a+bc = (a+b)(a+c)\]
Canonical Forms

• Two types:
  • Sum of Products (SOP)
  • Product of Sums (POS)

• Product of Sums:
  • a.k.a. conjunctive normal form, maxterm expansion
  • Maxterm: a sum (OR) involving all inputs
  • POS: Product (AND) maxterms for which the output is FALSE
  • Can obtain POSs from applying DeMorgan’s law to the SOPs of F (and vice versa)

<table>
<thead>
<tr>
<th>Maxterms</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
<th>f'</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b+c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a+b+c'</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a+b'+c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a+b'+c'</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a'+b+c</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a'+b+c'</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a'+b'+c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a'+b'+c'</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

One sum (or) term for each 0 in f:

\[ f = (a+b+c) (a+b+c') (a+b'+c) \]
\[ f' = (a+b'+c') (a'+b+c) (a'+b+c') (a'+b'+c) \]
Quiz

• Derive the product-of-sums form of $\bar{Y}$ based on the truth table.

a) $\bar{Y} = (A + B)(A + \bar{B})$

b) $\bar{Y} = A\bar{B} + AB$

c) $\bar{Y} = \bar{A}B + \bar{A}B$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$B$</td>
<td>$Y$</td>
<td>$\bar{Y}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Boolean Simplification
Example: Full Adder (FA) Carry out

\[ \text{co} = a'b'c + ab'c + abc' + abc \]
\[ = a'b'c + ab'c + abc' + abc + abc \]
\[ = a'b'c + abc + ab'c + abc' + abc \]
\[ = (a' + a)bc + ab'c + abc' + abc \]
\[ = (1)bc + ab'c + abc' + abc \]
\[ = bc + ab'c + abc' + abc + abc \]
\[ = bc + ab'c + abc + abc' + abc \]
\[ = bc + a(b' + b)c + abc' + abc \]
\[ = bc + a(1)c + abc' + abc \]
\[ = bc + ac + ab(c' + c) \]
\[ = bc + ac + ab(1) \]
\[ = bc + ac + ab \]
Why do Boolean simplification?

• Minimize number of gates in circuit
  • Gates take area

• Minimize amount of wiring in circuit
  • Wiring takes space and is difficult to route
  • Physical gates have limited number of inputs

• Minimize number of gate levels
  • Faster is better

• How to systematically simplify Boolean logics?
  • Use tools!
Practical methods for Boolean simplification

• Still based on Boolean algebra, but more systematic

• 2-level simplification -> multilevel

• Key tool: The Uniting Theorem

\[ a\bar{b} + ab = a (b\bar{b} + b) = a (1) = a \]

\[ f = ab\bar{b} + ab = a(b\bar{b}+b) = a \]

<table>
<thead>
<tr>
<th>ab</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
</tbody>
</table>
| 01 | 0  | b values change within rows
| 10 | 1  | a values don’t change
| 11 | 1  | b is eliminated, a remains
Karnaugh Map Method

- K-map is an alternative method of representing the truth table and to help visualize the adjacencies.

Note: “Gray code” labeling.
Karnaugh Map Method

- Adjacent groups of 1’s represent product terms

**OR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</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>

**Karnaugh Map**

\[ F = A + B \]
Karnaugh Map Method

1. Draw K-map of the appropriate number of variables.
2. Fill in map with function values from truth table.
3. Form groups of 1’s.
   ✓ Dimensions of groups must be even powers of two (1x1, 1x2, 1x4, ..., 2x2, 2x4, ...)
   ✓ Form as large as possible groups and as few groups as possible.
   ✓ Groups can overlap (this helps make larger groups)
   ✓ Remember K-map is periodical in all dimensions (groups can cross over edges of map and continue on other side)
4. For each group write a product term.
   • The term includes the “constant” variables (use the uncomplemented variable for a constant 1 and complemented variable for constant 0)
5. Form Boolean expression as sum-of-products.
Karnaugh Map Method

\[ F = A + B \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Product-of-Sums Version

1. Form groups of 0’s instead of 1’s.
2. For each group write a sum term.
   • the term includes the “constant” variables (use the uncomplemented variable for a constant 0 and complemented variable for constant 1)
3. Form Boolean expression as product-of-sums.

\[ f = (b' + c + d)(a' + c + d')(b + c + d') \]
Karnaugh Maps with Don’t Cares

- Don’t cares (x’s) in the truth table can be either 0’s or 1’s

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Administrivia

- Midterm 1: Thursday 10/5, 7-8:30pm
- Additional office hours (Thomas and William) on Wednesday and Thursday for homework help
- Lab 3 finished
  - If you couldn’t attend the lab, please get checked off in office hours this week
- Lab 4 is this week
  - Cycle wraps up on Monday
  - It is a moderately complex lab (~100 lines of verbose code), and relies on prior knowledge
  - Class ‘ambassadors’
  - Lab 5 starts next week
- Homework 3 due on Friday
Building a RISC-V Processor
Berkeley RISC-V ISA

www.riscv.org

- An open, license-free ISA
  - Runs GCC, LLVM, Linux distributions, ...
  - RV32, RV64, and RV128 variants for 32b, 64b, and 128b address spaces
- Originally developed for teaching classes at Berkeley, now widely adopted
- Base ISA only ~40 integer instructions
- Extensions provide full general-purpose ISA, including IEEE-754/2008 floating-point
- Designed for extension, customization
- Developed at UC Berkeley, now maintained by RISC-V International
- Open and commercial implementations
- RISC-V ISA, datapath, and control covered in CS61C; summarized here
RISC-V Processor Design

• Spec: Unprivileged ISA, RV32I (and a look at RV64I)

  Specification
  (e.g. in plain text)

  Model
  (e.g. in C/C++)

  Tests and
test vectors

  Architecture
  (e.g. in-order, out-of-order)

  RTL Logic Design
  (e.g. in Verilog)

Validation

Verification

• Tests provided as a part of the project

• Architecture: Single-cycle and pipelined in-order processor
  • Expanded from CS61C
One-Instruction-Per-Cycle RISC-V Machine

- On every tick of the clock, the computer executes one instruction.
- Current state outputs drive the inputs to the combinational logic, whose outputs settles at the values of the state before the next clock edge.
- At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle.
State Required by RV32I ISA

Each instruction reads and updates this state during execution:

- **Registers** ($x0 \ldots x31$)
  - Register file (*regfile*) $\text{Reg}$ holds 32 registers $\times$ 32 bits/register: $\text{Reg}[0] \ldots \text{Reg}[31]$
  - First register read specified by $rs1$ field in instruction
  - Second register read specified by $rs2$ field in instruction
  - Write register (destination) specified by $rd$ field in instruction
  - $x0$ is always 0 (writes to $\text{Reg}[0]$ are ignored)

- **Program counter** (PC)
  - Holds address of current instruction

- **Memory** (MEM)
  - Holds both instructions & data, in one 32-bit byte-addressed memory space
  - We’ll use separate memories for instructions (IMEM) and data (DMEM)
    - These are placeholders for instruction and data caches
  - Instructions are read (fetched) from instruction memory
  - Load/store instructions access data memory
Stages of the Datapath: Overview

• Problem: A single, “monolithic” CL block that “executes an instruction” (performs all necessary operations beginning with fetching the instruction and completing with the register access) is be too bulky and inefficient

• Solution: Break up the process of “executing an instruction” into stages, and then connect the stages to create the whole datapath
  — smaller stages are easier to design
  — easy to optimize (change) one stage without touching the others (modularity)
Five Stages of the Datapath

• Stage 1: Instruction Fetch (IF)

• Stage 2: Instruction Decode (ID)

• Stage 3: Execute (EX) - ALU (Arithmetic-Logic Unit)

• Stage 4: Memory Access (MEM)

• Stage 5: Write Back to Register (WB)
Basic Phases of Instruction Execution

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Register Write
Datapath Components: Combinational

- Combinational Elements

- Storage Elements + Clocking Methodology

- Building Blocks
Datapath Elements: State and Sequencing (1/4)

• Register

```
always @(posedge clk)
  if (wen) dataout <= datain;
endmodule
```

• Write Enable:
  • Negated (or deasserted) (0):
    Data Out will not change
  • Asserted (1): Data Out will become
    Data In on positive edge of clock
Datapath Elements: State and Sequencing (2/4)

- Register file (regfile, RF) consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
  - x0 is wired to 0

- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1

- Clock input (clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid $\Rightarrow$ busA or busB valid after “access time.”
Datapath Elements: State and Sequencing (3/4)

• Reg file in Verilog

```verilog
module rv32i_regs (  
    input clk, wen,  
    input [4:0] rw,  
    input [4:0] ra,  
    input [4:0] rb,  
    input [31:0] busw,  
    output [31:0] busa,  
    output [31:0] busb
);

    reg [31:0] reg_file [31:0];  
    always @(posedge clk)  
        if (wen) reg_file[rw] <= busw;  
    assign busa = (ra == 5’d0) ? 32’d0: regs[ra];  
    assign busb = (rb == 5’d0) ? 32’d0: regs[rb];
endmodule
```

• How does RV64I register file look like?

`x0` is zero

EECS151 L08 RISC-V
“Magical” memory

- One input bus: Data In
- One output bus: Data Out

Memory word is found by:

- For Read: Address selects the word to put on Data Out
- For Write: Set Write Enable = 1: address selects the memory word to be written via the Data In bus

Clock input (CLK)

- CLK input is a factor ONLY during write operation
- During read operation, behaves as a combinational logic block: Address valid $\Rightarrow$ Data Out valid after “access time”

Real memory later in the class
Review: Complete RV32I ISA

• Need datapath and control to implement these instructions
R-Format Instructions: Datapath
## Summary of RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>31 30 25 24 21 20 19 15 14 12 11 8 7 6 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
</tr>
<tr>
<td>imm[11:0]</td>
</tr>
<tr>
<td>imm[31:12]</td>
</tr>
</tbody>
</table>

- **R-type**
- **I-type**
- **S-type**
- **B-type**
- **U-type**
- **J-type**
R-Format Instruction Layout

- 32-bit instruction word divided into six fields of varying numbers of bits each: \(7 + 5 + 5 + 3 + 5 + 7 = 32\)

- Examples
  - **opcode** is a 7-bit field that lives in bits 6-0 of the instruction
  - **rs2** is a 5-bit field that lives in bits 24-20 of the instruction
R-Format Instructions opcode/funct fields

- **opcode**: partially specifies what instruction it is
  - Note: This field is equal to `0110011_\text{two}` for all R-Format register-register arithmetic instructions

- **funct7+funct3**: combined with opcode, these two fields describe what operation to perform

- **Question**: You have been professing simplicity, so why aren’t opcode and funct7 and funct3 a single 17-bit field?
  - Simpler implementation is more important than simpler spec
R-Format Instructions register specifiers

- **rs1** (Source Register #1): specifies register containing first operand
- **rs2**: specifies second register operand
- **rd** (Destination Register): specifies register which will receive result of computation
- Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (x0-x31)
R-Format Example

- RISC-V Assembly Instruction:
  \[
  \text{add} \quad x_{18}, x_{19}, x_{10}
  \]

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

\[
00000000 \quad 01010 \quad 10011 \quad 000 \quad 10010 \quad 0110011
\]

- add \quad rs2=10 \quad rs1=19 \quad \text{add} \quad \text{rd}=18 \quad \text{Reg-Reg OP}
Implementing the **add** instruction

The `add` instruction modifies the machine's state by:

- Adding the values in registers `rs1` and `rs2` and storing the result in register `rd`.
- Incrementing the program counter by 4.

**Example:**

- **opcode:** `0110011`
- **funct7, funct3, rd, rs1, rs2:** `0000000, 5, 5, 3, 7`
- **Description:**

```
add rs2 rs1 add rd Reg-Reg OP
```

```
add rd, rs1, rs2
```

**Instruction Details:**

- **Reg[rd] = Reg[rs1] + Reg[rs2]**
- **PC = PC + 4**
Datapath for \texttt{add}

\[ \text{PC} = \text{PC} + 4 \quad \text{Reg}[\text{rd}] = \text{Reg}[\text{rs1}] + \text{Reg}[\text{rs2}] \]
Timing Diagram for **add**

Clock

- **PC**: 1000
- **PC+4**: 1004
- **Inst[31:0]**: 
  - `add x1, x2, x3`
  - `add x6, x7, x9`
- **Reg[rs1]**
- **Reg[rs2]**
- **alu**
- **Reg[1]**
- **Reg[2]**
- **Reg[3]**
- **Reg[7]**
- **Reg[9]**

**IMEM**

- **AddrA**, **DataA**: Inst[24:20]
- **AddrB**, **DataB**: Inst[19:15]
- **AddrD**, **DataD**: Inst[11:7]

**ALU**

- **RegWEn**
- **Add**
- **addr**, **inst**
- **Reg[rs1]**, **Reg[rs2]**

**alu** + 4

**Reg[2] + Reg[3]**

**Reg[7] + Reg[9]**
Implementing the sub instruction

- sub rd, rs1, rs2
  - Almost the same as add, except now have to subtract operands instead of adding them
  - $\text{inst}[30]$ selects between add and subtract
Datapath for add/sub

- **Add**: clk, addr, inst
Implementing other R-Format instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs2</th>
<th>rs1</th>
<th>funct7</th>
<th>rd</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
</tr>
</tbody>
</table>

- All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function
Summary

• Boolean algebra
  • Deal with variables that are either True or False
  • Map naturally to hardware logic gates
  • Use theorems of Boolean algebra and Karnaugh maps to simplify equations

• RISC-V ISA
  • Open, with increasing adoption

• RISC-V processor
  • A large state machine
  • Datapath + control
  • Reviewed R-format instructions and corresponding datapath elements