Chapter Five

The Processor: Datapath & Control

• We’re ready to look at an implementation of the MIPS
• Simplified to contain only:
  – memory-reference instructions: lw, sw
  – arithmetic-logical instructions: add, sub, and, or, slt
  – control flow instructions: beq, j
• Generic Implementation:
  – use the program counter (PC) to supply instruction address
  – get the instruction from memory
  – read registers
  – use the instruction to decide exactly what to do
• All instructions use the ALU after reading the registers
5.1 Introduction

- The Five Classic Components of a Computer
- An abstract view of major functions of MIPS is shown in Fig. 5.1. Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

The Big Picture: The Performance Perspective

- Performance of a machine is determined by:
  - Instruction count
  - Clock cycle time
  - Clock cycles per instruction
- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- Today:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time
5.2 State Elements: Latches and Registers

- Unclocked (seldom used) vs. Clocked state elements
- Output is equal to the stored value inside the element (don't need to ask for permission to look at the value)
- Clocks are used in synchronous logic
  - when should an element that contains state be updated?
  - Change of state (value) is based on the clock
  - Latches: whenever the inputs change, and the clock is asserted
  - Flip-Flop: state changes only on a clock edge (edge-triggered methodology)

![Diagram of cycle time with rising and falling edges]

D-latch and D-register

- D-latch (6.3.3 of Digital Logic)
  - Two inputs:
    - the data value to be stored (D)
    - the clock signal (C) indicating when to read & store D
  - Two outputs:
    - the value of the internal state (Q) and its complement

![Diagram of D-latch]

- D-register: Output changes only on the clock edge (6.4.2)

![Diagram of D-register]

Tsung-Han Tsai
Register File

- Built using D flip-flops

Our Implementation

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements
5.2 Building a Datapath

- **Adder**
  - Carry In
  - Sum
  - Carry out
  - A \(32\)
  - B \(32\)
  - \(Y_{32}\)

- **MUX**
  - Select
  - A \(32\)
  - B \(32\)
  - \(Y_{32}\)
  - ALU control

- **ALU**
  - Result
  - A \(32\)
  - B \(32\)

- **N-bit Register**
  - Consist of \(N\) D Flip-Flop
  - \(N\)-bit input and output
  - Write Enable: asserted (1): Data Out will become Data In

Clocking Methodology

- All storage elements are clocked by the same clock edge
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew
- \((\text{CLK-to-Q} + \text{Shortest Delay Path} - \text{Clock Skew}) \ > \ \text{Hold Time}\)
**Storage Element: Idealized Memory**

- **Memory (idealized)**
  - One input bus: Data In
  - One output bus: Data Out

- **Memory word is selected by:**
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus

- **Address valid => Data Out valid after Access time.**

- Fig.5.4a show the abstraction of instruction memory and Fig.5.8a shows abstraction for data memory

```
Write Enable  Address
Data In   DataOut
32        32
```

**Storage Element: Register File**

- **Register File consists of 32 registers:**
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW

- **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)**
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid => busA or busB valid after Access time.

```
Write Enable  RWRARB  busW
32  32-bit Registers
32  32

busA  busB
32  32

Clk```

Tsung-Han Tsai
Some Simple Examples

- A portion of the datapath used for fetching INS and incrementing the PC: Fig. 5.5
- The datapath for R-type Ins: Fig. 5.7

Some Simple Examples

- The datapath for Load and Store Ins: Fig. 5.9
- The datapath for a branch operation: Fig. 5.10
5.3 A Simple Implementation Scheme

- Example: P351; Use multiplexors to select different data for an input of a block
- Fig.5.11, Fig.5.12 and Fig.5.13 show a step-by-step construction

ALU Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- Example:
  add $8, $17, $18
  Instruction Format:

<table>
<thead>
<tr>
<th>000000</th>
<th>10001</th>
<th>10010</th>
<th>01000</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>op  rs  rt  rd  shamt  funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- ALU's operation based on instruction type and function code
ALU Control

- Example: lw $1, 100($2)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
</tr>
</tbody>
</table>

- ALU control input

<table>
<thead>
<tr>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AND</td>
</tr>
<tr>
<td>001</td>
<td>OR</td>
</tr>
<tr>
<td>010</td>
<td>add</td>
</tr>
<tr>
<td>110</td>
<td>subtract</td>
</tr>
<tr>
<td>111</td>
<td>set-on-less-than</td>
</tr>
</tbody>
</table>

- Why is the code for subtract 110 and not 011?

Control

- Must describe hardware to compute 3-bit ALU control input
  - given instruction type
    - 00 = lw, sw
    - 01 = beq, 10 = arithmetic
  - function code for arithmetic

- Describe it using a truth table (can turn into gates):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUOp</th>
<th>Func field</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>ALUOp1</td>
<td>ALUOp0</td>
</tr>
<tr>
<td>LW or SW</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Branch Equal</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R-type(add)</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>R-type(subtract)</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>R-type(AND)</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>R-type(OR)</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>R-type(set on less)</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

Tsung-Han Tsai
Control

Control Circuits

- Simple combinational logic (truth tables)
Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path

The four steps of R-type Ins (Fig.5.21,22,23,24)
The five steps of lw Ins (Fig.5.25)

The three steps of beq-type Ins (Fig.5.26)
Single Cycle Implementation

• Calculate cycle time assuming negligible delays in line and other small blocks except:
  – memory (2ns), ALU and adders (2ns), register file (1ns)

Where we are headed: 5.4 Multicycle Approach

• Single Cycle Problems:
  – What if we had a more complicated instruction like floating point operation? ->The cycle time must consider worst case
  – wasteful of area, Example: one ALU + 2 adders

• Multicycle Solution:
  – use a “smaller” cycle time
    • break up the instructions into steps, each step takes a cycle
    • balance the amount of work to be done
    • have different instructions take different numbers of cycles
  – restrict each cycle to use only one major functional unit
    • At the end of a cycle stores values for use in later cycles (easiest thing to do)
    • A “multicycle datapath”: reusing functional units (1) ALU used to compute address and to increment PC (2) Memory used for instruction and data
    • Must introduce additional “internal” registers
Multicycle Approach (Registers shall be added)

- Our control signals will not be determined solely by instruction
- At the end of a clock cycle, all data that are used in subsequent clock cycles must be stored in register

Multicycle Approach (Multiplexor is added)
Multicycle with necessary controls

Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!
**Step 1: Instruction Fetch**

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

\[
\text{IR} = \text{Memory}[\text{PC}]; \\
\text{PC} = \text{PC} + 4;
\]

*Can we figure out the values of the control signals?*

*What is the advantage of updating the PC now?*

---

**Step 2: Instruction Decode and Register Fetch**

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

\[
\begin{align*}
A & = \text{Reg}[\text{IR}[25-21]]; \\
B & = \text{Reg}[\text{IR}[20-16]]; \\
\text{ALUOut} & = \text{PC} + (\text{sign-extend(\text{IR}[15-0])} \ll 2); \\
\end{align*}
\]

- We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)
Step 3 Execution (instruction dependent)

• ALU is performing one of three functions, based on instruction type
  – Memory Reference:
    \[ \text{ALUOut} = A + \text{sign-extend(IR}[15-0]) \];
  – R-type:
    \[ \text{ALUOut} = A \text{ op B} \];
  – Branch:
    \[ \text{if (A==B) PC = ALUOut} \];

Step 4 R-type or memory-access

• Loads and stores access memory
  \[ \text{MDR} = \text{Memory}[\text{ALUOut}] \];
  or
  \[ \text{Memory}[\text{ALUOut}] = B \];
• R-type instructions finish
  \[ \text{Reg}[\text{IR}[15-11]] = \text{ALUOut} \];

*The write actually takes place at the end of the cycle on the edge*
Step 5 Write-back step

- \( \text{Reg}[\text{IR}[20-16]] = \text{MDR}; \)

### Summary:

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>( \text{IR} = \text{Memory}[\text{PC}] )</td>
<td></td>
<td>( \text{PC} = \text{PC} + 4 )</td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td>( \text{A} = \text{Reg}[\text{IR}[25-21]] )</td>
<td>( \text{B} = \text{Reg}[\text{IR}[20-16]] )</td>
<td>( \text{ALUOut} = \text{PC} + (\text{sign-extend}(\text{IR}[15-0])) &lt;&lt; 2 )</td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/jump completion</td>
<td>( \text{ALUOut} = \text{A} \oplus \text{B} )</td>
<td>( \text{ALUOut} = \text{A} + \text{sign-extend}(\text{IR}[15-0]) )</td>
<td>If ( \text{A} = \text{B} ) then ( \text{PC} = \text{ALUOut} )</td>
<td>( \text{PC} = \text{PC}[31-28] \oplus (\text{IR}[25-0] &lt;&lt; 2) )</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>( \text{Reg}[\text{IR}[15-11]] = \text{ALUOut} )</td>
<td>Load: ( \text{MDR} = \text{Memory}[(\text{ALUOut}) ) or Store: ( \text{Memory}[(\text{ALUOut}) = \text{B} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: ( \text{Reg}[\text{IR}[20-16]] = \text{MDR} )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simple Questions

• How many cycles will it take to execute this code?

```
lw $t2, 0($t3)
lw $t3, 4($t3)
beq $t2, $t3, Label #assume not
add $t5, $t2, $t3
sw $t5, 8($t3)
Label: ...
```

• What is going on during the 8th cycle of execution?
• In what cycle does the actual addition of $t2$ and $t3$ take place?

Implementing the Control

• Value of control signals is dependent upon:
  – what instruction is being executed
  – which step is being performed

• Use the information we've accumulated to specify a finite state machine
  – specify the finite state machine graphically, or
  – use microprogramming

• Implementation can be derived from specification
Review: finite state machines

- Finite state machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)

- We'll use a Moore machine (output based only on current state)

Graphical Specification of FSM

- How many state bits will we need?
Finite State Machine for Control

- Implementation:

Programmable Logic Array (PLA) Implementation

- If I picked a horizontal or vertical line could you explain it?

For example:

\[ PCWrite = s_0s_1s_2s_3 + s_0s_1s_2s_3 \]
ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is \( m \)-bits, we can address \( 2^m \) entries in the ROM.
  - our outputs are the bits of data that the address points to.

\[
\begin{array}{|c|c|c|}
\hline
m & n & \text{Address Data} \\
\hline
\hline
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
\hline
\end{array}
\]

\( m \) is the "height", and \( n \) is the "width"

How many inputs are there?
6 bits for opcode, 4 bits for state = 10 address lines
(i.e., \( 2^{10} = 1024 \) different addresses)

How many outputs are there?
16 datapath-control outputs, 4 state bits = 20 outputs

ROM is \( 2^{10} \times 20 = 20K \) bits (and a rather unusual size)

Rather wasteful, since for lots of the entries, the outputs are the same
— i.e., opcode is often ignored
ROM vs PLA

- Break up the table into two parts
  - 4 state bits tell you the 16 outputs, $2^4 \times 16$ bits of ROM
  - 10 bits tell you the 4 next state bits, $2^{10} \times 4$ bits of ROM
  - Total: 4.3K bits of ROM

- PLA is much smaller
  - can share product terms
  - only need entries that produce an active output
  - can take into account don't cares

- Size is $(\#\text{inputs} \times \#\text{product-terms}) + (\#\text{outputs} - \#\text{product-terms})$
  
  For this example = $(10 \times 17) + (20 \times 17) = 460$ PLA cells

- PLA cells usually about the size of a ROM cell
  (slightly bigger)

5.5 Microprogramming: Simplifying Control Design

- Control is the hard part of processor design
  - Datapath is fairly regular and well-organized
  - Memory is highly regular
  - Control is irregular and global

- Microprogramming: Designing the control as a program that implements the machine instructions in terms of simpler microinstructions at the level of register transfer operations
  - Think of the control operation as to issue set of control signals (in sequence) that must be asserted in a state as an “microinstruction” to be executed by the datapath.
  - Executing a microinstruction has the effect of asserting the control signals specified by the microinstruction

- A microinstruction is represented as a sequence of fields whose functions are related (example: Fig. 5.44)
  - Usually placed in a ROM or a PLA (have address)
  - Three methods to choose next microINS: P.401

- A microprogram is a symbolic representation of the control that will be translated by a program to control logic
### Microinstruction format

<table>
<thead>
<tr>
<th>Field name</th>
<th>Value</th>
<th>Signals active</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>ALUOp = 00</td>
<td>Cause the ALU to add</td>
<td></td>
</tr>
<tr>
<td>Sub</td>
<td>ALUOp = 01</td>
<td>Cause the ALU to subtract; this implements the compare for branches</td>
<td></td>
</tr>
<tr>
<td>Func code</td>
<td>ALUOp = 10</td>
<td>Use the instruction’s function code to determine ALU control</td>
<td></td>
</tr>
<tr>
<td>SRC1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>ALUSrcA = 0</td>
<td>Use the PC as the first ALU input</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>ALUSrcB = 1</td>
<td>Register B is the first ALU input</td>
<td></td>
</tr>
<tr>
<td>SRC2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>ALUSrcB = 00</td>
<td>Use the second ALU input</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ALUSrcB = 10</td>
<td>Use output of the sign extension unit as the second ALU input</td>
<td></td>
</tr>
<tr>
<td>Extshft</td>
<td>ALUSrcB = 11</td>
<td>Use the output of the shift-by-two unit as the second ALU input</td>
<td></td>
</tr>
<tr>
<td>Register control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write ALU</td>
<td>RegWrite, RegDst = 1, MemtoReg = 0</td>
<td>Write a register using the rt field of the IR as the register number and the contents of the ALUOut as the data</td>
<td></td>
</tr>
<tr>
<td>Write MDR</td>
<td>RegWrite, RegDst = 0, MemtoReg = 1</td>
<td>Write a register using the rt field of the IR as the register number and the contents of the MDR as the data</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read PC</td>
<td>MemRead, b0 = 0</td>
<td>Read memory using the PC as address; write result into IR (and the MDR)</td>
<td></td>
</tr>
<tr>
<td>Read ALU</td>
<td>MemRead, b0 = 1</td>
<td>Read memory using the ALUOut as address; write result into MDR</td>
<td></td>
</tr>
<tr>
<td>Write ALU</td>
<td>MemWrite, b0 = 1</td>
<td>Write memory using the ALUOut as address, contents of B as the data</td>
<td></td>
</tr>
<tr>
<td>PC write control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>PSource = 00, PChWrite</td>
<td>Write the output of the ALU into the PC</td>
<td></td>
</tr>
<tr>
<td>ALUOut-cond</td>
<td>PSource = 01, PChWrite</td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut</td>
<td></td>
</tr>
<tr>
<td>Jump address</td>
<td>PSource = 10, PChWrite</td>
<td>Write the PC with the jump address from the instruction</td>
<td></td>
</tr>
<tr>
<td>Sequencing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq</td>
<td>AddrSel = 11</td>
<td>Choose the next microinstruction sequentially</td>
<td></td>
</tr>
<tr>
<td>Fetch</td>
<td>AddrSel = 00</td>
<td>Go to the first microinstruction to begin a new instruction</td>
<td></td>
</tr>
<tr>
<td>Dispatch 1</td>
<td>AddrSel = 01</td>
<td>Dispatch using the ROM 1</td>
<td></td>
</tr>
<tr>
<td>Dispatch 2</td>
<td>AddrSel = 10</td>
<td>Dispatch using the ROM 2</td>
<td></td>
</tr>
</tbody>
</table>

### Microprogramming

```
control unit
microcode memory

outputs

input

adder

microprogram counter

address select logic

instruction register

op code field
```

Tsung-Han Tsai

48
Another Implementation Style

- Complex instructions: the "next state" is often current state + 1

---

Details

<table>
<thead>
<tr>
<th>Dispatch ROM 1</th>
<th>Op</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>R</td>
<td>R-format</td>
<td>0110</td>
</tr>
<tr>
<td>000110</td>
<td>J</td>
<td>Jbn</td>
<td>1001</td>
</tr>
<tr>
<td>001010</td>
<td>I</td>
<td>Iw</td>
<td>0000</td>
</tr>
<tr>
<td>100011</td>
<td>LW</td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td>101011</td>
<td>SW</td>
<td></td>
<td>0010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dispatch ROM 2</th>
<th>Op</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100111</td>
<td>J</td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td>101011</td>
<td>SW</td>
<td></td>
<td>0101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State number</th>
<th>Address-control action</th>
<th>Value of AddrCtl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Use dispatch ROM 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Use dispatch ROM 2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
</tbody>
</table>
Maximally vs. Minimally Encoded

• No encoding:
  – 1 bit for each datapath operation
  – faster, requires more memory (logic)
  – used for Vax 780 — an astonishing 400K of memory!
• Lots of encoding:
  – send the microinstructions through logic to get control signals
  – uses less memory, slower
• Historical context of CISC:
  – Too much logic to put on a single chip with everything else
  – Use a ROM (or even RAM) to hold the microcode
  – It’s easy to add new instructions

Microcode: Trade-offs

• Specification Advantages:
  – Easy to design and write
  – Design architecture and microcode in parallel
• Implementation (off-chip ROM) Advantages
  – Easy to change since values are in memory
  – Can emulate other architectures
  – Can make use of internal registers
• Implementation Disadvantages, SLOWER now that:
  – Control is implemented on same chip as processor
  – ROM is no longer faster than RAM
  – No need to go back and make changes
### The Big Picture

<table>
<thead>
<tr>
<th>Initial representation</th>
<th>Finite state diagram</th>
<th>Microprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequencing control</td>
<td>Explicit next state function</td>
<td>Microprogram counter + dispatch ROMS</td>
</tr>
<tr>
<td>Logic representation</td>
<td>Logic equations</td>
<td>Truth tables</td>
</tr>
<tr>
<td>Implementation technique</td>
<td>Programmable logic array</td>
<td>Read only memory</td>
</tr>
</tbody>
</table>

Tsung-Han Tsai