Systems Architecture I

Topics
Review of Digital Circuits and Logic Design
Review of Sequential Logic Circuits
Compilers, Assemblers, Linkers & Loaders

Notes Courtesy of Jeremy R. Johnson
Systems Architecture I

Topic 1: Review of Digital Circuits and Logic Design
Introduction

• Objective: To understand how the simple model computer from the previous lecture could be implemented using logic gates.

• Review of Boolean functions and expressions
• Review of logic gates
• Decoders, Encoders, and Multiplexors

References: Dewdney, The New Turing Omnibus (Chapter 3, 13, and 28) and Sec. B1-B3 of the text.
A Boolean variable has two possible values (true/false) (1/0).

A Boolean function has a number of Boolean input variables and has a Boolean valued output.

A Boolean function can be described using a truth table.

There are $2^n$ Boolean functions of $n$ variables.

**Multiplexor function**

<table>
<thead>
<tr>
<th>s</th>
<th>x₀</th>
<th>x₁</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>0</td>
<td>1</td>
<td>0</td>
<td>0</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>0</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Boolean Expressions

- An expression built up from variables, and, or, and not.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| x | y | x•y | x | y | x+y | x | x | x
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1

and or not
Boolean Expressions

- A Boolean expression is a Boolean function.

- Any Boolean function can be written as a Boolean expression
  - Disjunctive normal form (sums of products)
  - For each row in the truth table where the output is true, write a product such that the corresponding input is the only input combination that is true
  - Not unique

- E.G. (multiplexor function)

\[ \overline{s} \cdot x_0 \cdot \overline{x}_1 + \overline{s} \cdot x_0 \cdot x_1 + s \cdot \overline{x}_0 \cdot x_1 + s \cdot x_0 \cdot x_1 \]
**Boolean Logic**

- Boolean expressions can be simplified using rules of Boolean logic
  - Identity law: \( A + 0 = A \) and \( A \cdot 1 = A \).
  - Zero and One laws: \( A + 1 = 1 \) and \( A \cdot 0 = 0 \).
  - Inverse laws: \( A + \overline{A} = 1 \) and \( A \cdot \overline{A} = 0 \).
  - Commutative laws: \( A + B = B + A \) and \( A \cdot B = B \cdot A \).
  - Associative laws: \( A + (B + C) = (A + B) + C \) and \( A \cdot (B \cdot C) = (A \cdot B) \cdot C \).
  - Distributive laws: \( A \cdot (B + C) = (A \cdot B) + (A \cdot C) \) and
    \[
    A + (B \cdot C) = (A + B) \cdot (A + C)
    \]
  - DeMorgan’s laws: \( A + B = \overline{A \cdot B} \) and \( A \cdot B = \overline{A + B} \).

- The reason for simplifying is to obtain shorter expressions, which we will see leads to simpler logic circuits.
Simplification of Boolean Expressions

- Simplifying multiplexor expression using Boolean algebra

\[ \overline{s} \cdot x_0 \cdot \overline{x}_1 + \overline{s} \cdot x_0 \cdot x_1 + s \cdot \overline{x}_0 \cdot x_1 + s \cdot x_0 \cdot x_1 \]

\[ = \overline{s} \cdot x_0 \cdot \overline{x}_1 + \overline{s} \cdot x_0 \cdot x_1 + s \cdot x_1 \cdot \overline{x}_0 + s \cdot x_1 \cdot x_0 \quad \text{(commutative law)} \]

\[ = \overline{s} \cdot x_0 \cdot (x_1 + \overline{x}_1) + s \cdot x_1 \cdot (x_0 + \overline{x}_0) \quad \text{(distributive law)} \]

\[ = \overline{s} \cdot x_0 \cdot 1 + s \cdot x_1 \cdot 1 \quad \text{(inverse law)} \]

\[ = \overline{s} \cdot x_0 + s \cdot x_1 \quad \text{(identity law)} \]

- Verify that the Boolean function corresponding to this expression has the same truth table as the original function.
Logic Circuits

• A single line labeled $\times$ is a logic circuit. One end is the input and the other is the output. If $A$ and $B$ are logic circuits so are:

  • and gate

  • or gate

  • inverter (not)
Logic Circuits

- Given a Boolean expression it is easy to write down the corresponding logic circuit
- Here is the circuit for the original multiplexor expression
Logic Circuits

- Here is the circuit for the simplified multiplexor expression
Nand Gates

• A nand gate is an inverted and gate

\[
\begin{array}{ccc}
  x & y & x \mid y \\
  0 & 0 & 1 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 0 \\
\end{array}
\]

\text{nand}

• All Boolean functions can be implemented using nand gates (and and not can be implemented using nand)
A decoder is a logic circuit that has $n$ inputs (think of this as a binary number) and $2^n$ outputs. The output corresponding to the binary input is set to 1 and all other outputs are set to 0.
Encoder

• An encoder is the opposite of a decoder. It is a logic circuit that has $2^n$ inputs and $n$ outputs. The output equal to the input line (in binary) that is set to 1 is set to 1.
A multiplexor is a switch which routes n inputs to one output. The input is selected using a decoder.
Implementing Logic Gates with Transistors

A Transistor NOT Gate

A Transistor NAND Gate
Exercises

• Prove De Morgan’s laws.

• Conjunctive normal form consists of products of sums. Obtain a conjunctive normal form for the multiplexor on slide 5 and draw the corresponding circuit. How does the number of gates compare with the circuit on slide 9.

• Design a $3 \times 8$ decoder.

• Design an $8 \times 3$ encoder.

• Redesign the multiplexor on slide 14 using only inverters, three-input NAND gates, and a single four-input NAND gate.

• Show a transistor NOR gate.
Systems Architecture I

Topic 2: Review of Sequential Logic Circuits
Introduction

• Objective: To understand how data can be stored in a computer.
  – Sequential vs. Combinational Logic
  – Flip-flop
  – Timed flip-flop
  – Implementing computer memory
  – Review of the simple computer model

References: Dewdney, The New Turing Omnibus (Chapter 38 and 48) and Sec. B4-B7 of the text.
Combinational vs. Sequential Circuits

• A combinational circuit is a logic circuit without any loops. The same outputs are always computed for the same inputs.

• A sequential circuit contains two-state memory elements which can remember the state over time. The outputs depend on both the inputs and the current state. The state changes over time which is marked off in discrete steps by pulses emanating from a clock. The pulses coordinate activity.
Flip-Flop (SR-Latch)

- A sequential logic circuit with two states.
- The two states are \((Q=0, Q'=1)\) and \((Q=1, Q'=0)\)
- Provided the input \((R = 0, S = 0)\) is not allowed, the flip-flop can only be in one of these two states.
Flip-Flop States

- The state (Q=0, Q’=1) corresponds to storing a 0.
- The state (Q=1, Q’=0) corresponds to storing a 1.
- If S (set) is 1, then the state is set to 1.
- If R (reset) is 1, then the state is set to 0.
- If R & S are 1, the state does not change.

<table>
<thead>
<tr>
<th>Old Q</th>
<th>R</th>
<th>S</th>
<th>Q’</th>
<th>Next Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ Q \]
\[ Q’ \]
\[ R \]
\[ S \]
Flip-Flop States

- The transitions of the flip-flop are conveniently described in the following state transition diagram (finite state machine). The arcs are labeled by the RS inputs that cause the transition.
- In the diagram, state 0 is when Q = 0 and Q' = 1, and state 1 is when Q = 1 and Q' = 0.
A Clocked Flip-Flop
N-bit Register

- A state variable containing N-bits
- Built from an array of N flip-flops
- State changes when load selected and the clock is high
N-bit Register

X_1

X_2

load

clock

R  Q
S  Q'

R  Q
S  Q'
Memory Cell

read = 1/write = 0

input

output

select
• \( N = 2^k \) words each \( l \) bits
• Inputs
  – \( k \) address bits
  – \( l \) data-in bits (for write)
  – flag to select read/write
• Outputs
  – \( l \) data-out bits (for read)

• Built from \( N \times l \) array of memory cells
  – input, output, flag, select
• Decoder used to decode address
Systems Architecture I

Topic 3: Compilers, Assemblers, Linkers & Loaders
Introduction

- Objective: To introduce the role of compilers, assemblers, linkers and loaders. To see what is underneath a C program: assembly language, machine language, and executable.
Below Your Program

Example from a Unix system

- **Source Files:** `count.c` and `main.c`
- **Corresponding assembly code:** `count.s` and `main.s`
- **Corresponding machine code (object code):** `count.o` and `main.o`
- **Library functions:** `libc.a`
- **Executable file:** `a.out`

- **format for `a.out` and object code:**
  ELF (Executable and Linking Format)
Producing an Executable Program

Example from a Unix system (SGI Challenge running IRIX 6.5)

• Compiler: count.c and main.c $\rightarrow$ count.s and main.s
  – gcc -S count.c main.c

• Assembler: count.s and main.s $\rightarrow$ count.o and main.o
  – gcc -c count.s main.s
  – as count.s -o count.o

• Linker/Loader: count.o main.o libc.a $\rightarrow$ a.out
  – gcc main.o count.o
  – ld main.o count.o -lc (additional libraries are required)
void main()
{
    int n,s;
    printf("Enter upper limit: ");
    scanf("%d",&n);
    s = count(n);
    printf("Sum of i from 1 to %d = %d\n",n,s);
}

int count(int n)
{
    int i,s;
    s = 0;
    for (i=1;i<=n;i++)
    {
        s = s + i;
    }
    return s;
}
Assembly Code for MIPS (count.s)

.#file 1 "count.c"
    .option pic2
    .section .text
    .text
    .align 2
    .globl count
    .ent count

count:
    .LFB1:
      .frame $fp,48,$31  # vars= 16, regs= 2/0, args= 0, extra= 1
      6
      .mask 0x50000000,-8
      .fmask 0x00000000,0
      subu $sp,$sp,48
    .LCFI0:
      sd $fp,40($sp)
.LCFI1:
    sd $28,32($sp)
.LCFI2:
    move $fp,$sp
.LCFI3:
    .set noat
    lui $1,%hi(%neg(%gp_rel(count)))
    addiu $1,$1,%lo(%neg(%gp_rel(count)))
    daddu $gp,$1,$25
    .set at
    sw $4,16($fp)
    sw $0,24($fp)
    li $2,1       # 0x1
    sw $2,20($fp)
.L3:
    lw $2,20($fp)
    lw $3,16($fp)
    slt $2,$3,$2
    beq $2,$0,.L6
    b .L4
.L6:
    lw $2,24($fp)
    lw $3,20($fp)
    addu $2,$2,$3
    sw $2,24($fp)
.L5:
    lw $2,20($fp)
    addu $3,$2,1
    sw $3,20($fp)
    b .L3
.L4:
    lw $3,24($fp)
    move $2,$3
    b .L2
.L2:
    move $sp,$fp
    ld $fp,40($sp)
    ld $28,32($sp)
    addu $sp,$sp,48
    j $31
.LFE1:
    .end      count
Executable Program for MIPS (a.out)

0000000 7f45 4c46 0102 0100 0000 0000 0000 0000
0000020 0002 0008 0000 0001 1000 1060 0000 0034
0000040 0000 6c94 2000 0024 0034 0020 0007 0028
0000060 0023 0022 0000 0006 0000 0034 1000 0034
0000100 1000 0034 0000 00e0 0000 00e0 0000 0004
0000120 0000 0004 0000 0003 0000 0114 1000 0114
0000140 1000 0114 0000 0015 0000 0015 0000 0004
0000160 0000 0001 7000 0002 0000 0130 1000 0130
0000200 1000 0130 0000 0080 0000 0080 0000 0004
0000220 0000 0008 7000 0000 0000 01b0 1000 01b0
0000240 1000 01b0 0000 0018 0000 0018 0000 0004
0000260 0000 0004 0000 0002 0000 01c8 1000 01c8
0000300 1000 01c8 0000 0108 0000 0108 0000 0004
0000320 0000 0004 0000 0001 0000 0000 1000 0000
0000340 1000 0000 0000 3000 0000 3000 0000 0005

........................