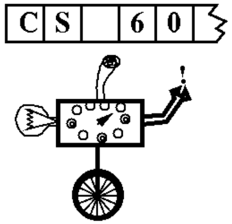
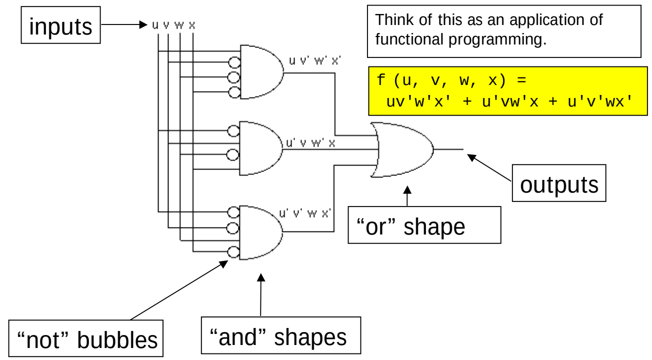


Logic Synthesis

November 9, 2001

Synthesizing Switching Functions

A "logic circuit" is composed of switching functions



SOP and Minterm forms

`f (u,v,w,x) = uv'w'x' + u'vw'x + u'v'wx'`

- This is called a **SOP** (sum-of-products) form. In this case, a "product" means product of variables or complemented variables.
- It is also a **minterm** form. A minterm is a product that uses ***all*** of the variables.
- Not every SOP is a minterm form.

Truth Table

```
f (u, v, w, x) =
        u v'w'x'
      + u'v w'x
      + u'v'w x'
```

u v w x	f
0 0 0 0	0
0 0 0 1	0
0 0 1 0	1
0 0 1 1	0
0 1 0 0	0
0 1 0 1	1
0 1 1 0	0
0 1 1 1	0
1 0 0 0	1
1 0 0 1	0
1 0 1 0	0
1 0 1 1	0
1 1 0 0	0
1 1 0 1	0
1 1 1 0	0
1 1 1 1	0

Note the Connection

```
f (u, v, w, x) =
        u v'w'x'
      + u'v w'x
      + u'v'w x'
```

u	v	w	x	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

The “1” rows of the truth table correspond exactly to the minterms!

Shorthands

```
f (u, v, w, x) =
        u v'w'x'
      + u'v w'x
      + u'v'w x'
```

Show only the “1” rows (be careful)

u	v	w	x	f
0	0	1	0	1
0	1	0	1	1
1	0	0	0	1

Represent whole table by a set of “minterm numbers”:

{2, 5, 8}

Represent whole table by a single numeral: 0010010010000000

Ways to Specify Boolean Functions

- Logic circuit
- Functional expression
  - SOP form
    - Minterm expansion
- Truth table
- Compressed truth table
- Index number
- Karnaugh map

Counting Functions

- The following are all equal:
  - The number of boolean functions of n variables.
  - The number of ways to assign 0 or 1 to the 2^n rows of the truth table.
  - The number of subsets of $\{0, 1, 2, \ldots, 2^n - 1\}$

Number of Switching Functions

- n = 1: 4
- n = 2: 16
- n = 3: 256
- n = 4: 65,536
- n = 5: 4,294,967,296
- n = 6: 18,446,744,073,709,551,616

Each level **squares** the previous.

The 16 switching functions of 2 variables

args		Function number															
b	**c**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	**0**	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	**1**	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	**0**	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	**1**	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

and **xor** **or** **nor** **equiv** **implies** **nand**

Logic Synthesis:
Abstraction to Implementation

- **From:** Verbal problem description
- **To:** Implementation as a network of basic switching functions

Logic Synthesis: Stages

1 **Provide** verbal problem description.
2 **Tabulate** description as function on finite sets.
3 **Encode** finite sets into bits.
4 **Transcribe** the encoded tables.
5 **Split** into individual switching functions.
6 **Realize** as a network of basic gates.

Example

- Verbal description:

  Implement a “mod 3 adder using logic gates”

- A definition of mod3 addition:

  `f(a, b) = (a+b)%3`

  where $a,b \in \{0,1,2\}$

Tabulate definition of function

(x+y)%3	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Encode sets into bits

- Set to be encoded: $\{0,1,2\}$

- Chosen encoding (among many):

  $0 \rightarrow 00$
  $1 \rightarrow 01$
  $2 \rightarrow 10$

Transcribe the Function to the Encoded Values

Function

(x+y)%3	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Encoding

$0 \rightarrow 00$
$1 \rightarrow 01$
$2 \rightarrow 10$

Encoded Function

Here first argument becomes uv, second becomes xy.

	uv	00	01	10
xy	00	00	01	10
	01	01	10	00
	10	10	00	01

Split the Encoded Function into individual **switching** functions

Encoded Function

xy \ uv	00	01	10
00	0 0	0 1	1 0
01	0 1	1 0	0 0
10	1 0	0 0	0 1

left bit

right bit

Call this f_1.

xy \ uv	00	01	10
00	0	0	1
01	0	1	0
10	1	0	0

Call this f_2.

xy \ uv	00	01	10
00	0	1	0
01	1	0	0
10	0	0	1

The resulting switching functions generally will only be ***partially*** specified.

f_1

xy \ uv	00	01	10
00	0	0	1
01	0	1	0
10	1	0	0

f_2

xy \ uv	00	01	10
00	0	1	0
01	1	0	0
10	0	0	1

u v w x	f_1 f_2
0 0 0 0	0 0
0 0 0 1	0 1
0 0 1 0	1 0
0 0 1 1	? ?
0 1 0 0	0 1
0 1 0 1	1 0
0 1 1 0	0 0
0 1 1 1	? ?
1 0 0 0	1 0
1 0 0 1	0 0
1 0 1 0	0 1
1 0 1 1	? ?
1 1 0 0	? ?
1 1 0 1	? ?
1 1 1 0	? ?
1 1 1 1	? ?

As the unspecified values will never occur, we can give the function either value 0 or 1.

For the time being, let's just make them all 0.

Now we can "read off" an expression for each function.

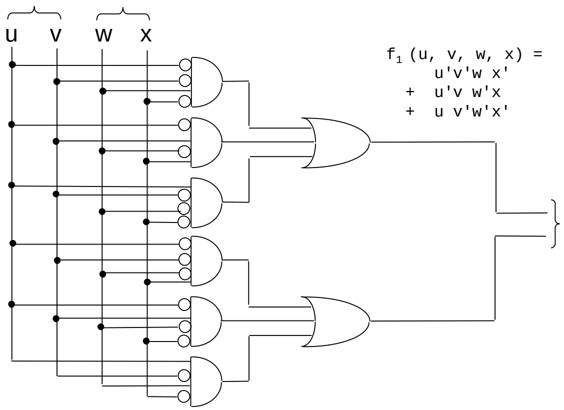
```
f1 (u, v, w, x) =
          u'v'w x
      +   u'v w'x
      +   u v'w'x'
```

u v w x	f_1 f_2
0 0 0 0	0 0
0 0 0 1	0 1
0 0 1 0	1 0
0 0 1 1	? ?
0 1 0 0	0 1
0 1 0 1	1 0
0 1 1 0	0 0
0 1 1 1	? ?
1 0 0 0	1 0
1 0 0 1	0 0
1 0 1 0	0 1
1 0 1 1	? ?
1 1 0 0	? ?
1 1 0 1	? ?
1 1 1 0	? ?
1 1 1 1	? ?

Exercise: "read off" the expression for f_2.

u v w x	f_1 f_2
0 0 0 0	0 0
0 0 0 1	0 1
0 0 1 0	1 0
0 0 1 1	? ?
0 1 0 0	0 1
0 1 0 1	1 0
0 1 1 0	0 0
0 1 1 1	? ?
1 0 0 0	1 0
1 0 0 1	0 0
1 0 1 0	0 1
1 0 1 1	? ?
1 1 0 0	? ?
1 1 0 1	? ?
1 1 1 0	? ?
1 1 1 1	? ?

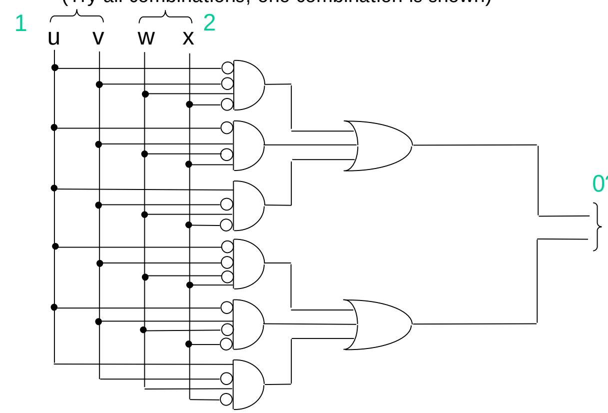
Realize each function by gates




```
f1 (u, v, w, x) =
    u'v'w x'
  + u'v w'x
  + u v'w'x'
```

Check by Simulating Circuit

(Try all combinations; one combination is shown)



Rex Implementation

```
encode(0) => [0, 0];
encode(1) => [0, 1];
encode(2) => [1, 0];

decode([0, 0]) => 0;
decode([0, 1]) => 1;
decode([1, 0]) => 2;
decode([x, y]) => "error, should not occur";

add3(i, j) = (i+j)%3;


addByCircuit(i, j) = decode(circuit(encode(i), encode(j)));

circuit([u, v], [w, x]) =>
    [   !u && !v &&  w && !x
     || !u &&  v && !w &&  x
     ||  u && !v && !w && !x,

        !u && !v && !w &&  x
     || !u &&  v && !w && !x
     ||  u && !v &&  w && !x];
```

Testing Code

```
test(addByCircuit(0, 0), add3(0, 0));
test(addByCircuit(0, 1), add3(0, 1));
test(addByCircuit(0, 2), add3(0, 2));

test(addByCircuit(1, 0), add3(1, 0));
test(addByCircuit(1, 1), add3(1, 1));
test(addByCircuit(1, 2), add3(1, 2));

test(addByCircuit(2, 0), add3(2, 0));
test(addByCircuit(2, 1), add3(2, 1));
test(addByCircuit(2, 2), add3(2, 2));
```