Lecture A2: X-TOY Programming and Simulation

DEC PDP 12

Useful X-TOY Idioms

Jump absolute.

- Jump to a fixed memory address.
  - branch if zero with destination
  - register 0

Jump absolute
17: C014 pc ← 14

Register assignment.

- No instruction that transfers contents of one register into another.
- Pseudo-instruction that simulates assignment:
  - add with register 0 as one of two source registers

Register assignment
17: 1230 R[2] ← R[3]

No-op.

- Instruction that does nothing.
- Plays the role of whitespace in C programs.
  - numerous other possibilities!

No-op
17: 1000 no-op

How to represent negative integers

X-TOY integers occupy 16 bits each.

- We can represent 0 to 2^{16}-1 if we like.
- But what about negative integers?
- Reserving half the possible bit-patterns for negative seems fair.

Highly desirable property:

- If X is a positive integer, then the representation of -X, when added to X, had better yield zero.

```
   X      0 0 1 1 0 1 0 0
+(-X)   + ? ? ? ? ? ? ? ?
   0      0 0 0 0 0 0 0 0

   X      0 0 1 1 0 1 0 0
          + 1 1 0 0 1 0 1 1
+(-X)     1 1 1 1 1 1 1 1
        +               1
   0      0 0 0 0 0 0 0 0
```


Two's Complement Integers

Properties:

- Leading bit (bit 15) signifies sign.
- Negative integer -N represented by 2^{16} - N.
- Trick to compute -N:

1. Start with N.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

2. Flip bits.

	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1

3. Add 1.

-4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0

Two's Complement Integers

		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Dec	Hex	Binary															
+32767	7FFF	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		• • •															
+4	0004	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
+3	0003	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
+2	0002	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
+1	0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
+0	0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-1	FFFF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-2	FFFE	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
-3	FFFD	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
-4	FFFC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
		• • •															
-32768	8000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Properties of Two's Complement Integers

Nice properties:

- 0000000000000000 represents 0.
- -0 and +0 are the same.
- Addition is easy (see next slide).
- Checking for arithmetic overflow is easy.

- N = ~N + 1

Not-so-nice properties.

- Can represent one more negative integer than positive integer ($-32{,}768 = -2^{15}$ but not $32{,}768 = 2^{15}$).

Two's Complement Arithmetic

Addition is carried out as if all integers were positive.

- It usually works:

-3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
+																
4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
=																
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Two's Complement Arithmetic

Addition is carried out as if all integers were positive.

- It usually works.
- But overflow can occur:

+32,767	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
+																
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
=																
-32,767	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Representing Other Primitive Data Types

Negative integers.

- X-TOY uses two's complement integers.

Big integers.

- Can use "multiple precision."
- Use two 16-bit words per integer.

Real numbers.

- Can use "floating point" (like scientific notation).
- Double word for extra precision.

Characters.

- Can use ASCII code (8 bits / character).
- Can pack two characters into one 16-bit word.

Memory Indirection

Static addressing.

- Until now, all load/store addresses hardwired in instruction.
- Ex. `8A34: R[A] ← mem[34]`
- More flexibility needed to implement arrays.

Indirect (dynamic) addressing.

- Work with names of data.
- Want to access variable memory location like x, instead of hardwiring 34.

Solution.

- Put memory address in register. (C "pointer")
- Use CONTENTS of register as address.
- Use store indirect, load indirect instructions to access.

array3.c

```
int i = 0;
int N = 33;

while (i < N) {
   a[i] += 3;
   i++;
}
```


Array3

array3.toy

```
10: 7101   R[1] ← 0001          constant 1
11: 7202   R[2] ← 0003          constant 3
12: 7A20   R[A] ← 0020          a[]
13: 7300   R[4] ← 0033          N ← 33
14: 7500   R[5] ← 0000          i ← 0

15: 17A5   R[7] ← R[A] + R[5]   address of a[i]
16: A907   R[9] ← mem[R[7]]     t ← a[i]
17: 1993   R[9] ← R[9] + R[2]   t += 3
18: B907   mem[[R[7]] ← R[9]    a[i] ← t
19: 1551   R[5] ← R[5] + R[1]   i++
1A: 2C45   R[C] ← R[4] - R[5]   N - i
1B: DC15   if (R[C] > 0) goto 15
1C: 0000   halt
```


Standard Input, Standard Output

Standard input.

- Loading from memory address FF loads one (hexadecimal) integer from X-TOY stdin.
- 8AFF means:
  - read an integer from standard input, and store it in register A
  - `scanf("%hX", &a);`

stdin-stdout.toy

```
10: 8AFF   read R[A]
11: 9AFF   write R[A]
12: 0000   halt
```

Standard output.

- Writing to memory location FF.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1
8_{16}				A_{16}				F_{16}				F_{16}			
opcode				dest d				addr							

Standard Input, Standard Output

Enables computer to process more information than fits in memory.

- Arbitrary amounts of input.

max.c

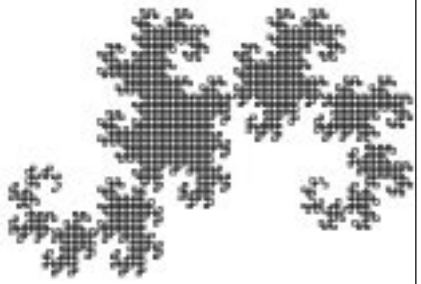
```
int x, max = 0;

while (scanf("%d", &x) != EOF) {
   if (x > max)
      max = x;
}

printf("Maximum = %d\n", max);
```

- Arbitrary amounts of output.
  - dragon curve

Bitwise Operations

Bitwise AND. (opcode 3)

Bitwise XOR. (opcode 4)



dragon-nonrecursive.c

```
void dragon(int m) {
   int k;
   F();
   for (k = 1; k <= m-1; k++) {
      if (k & ((k^(k-1))+1))
         R();
      else
         L();
      F();
   }
}
```

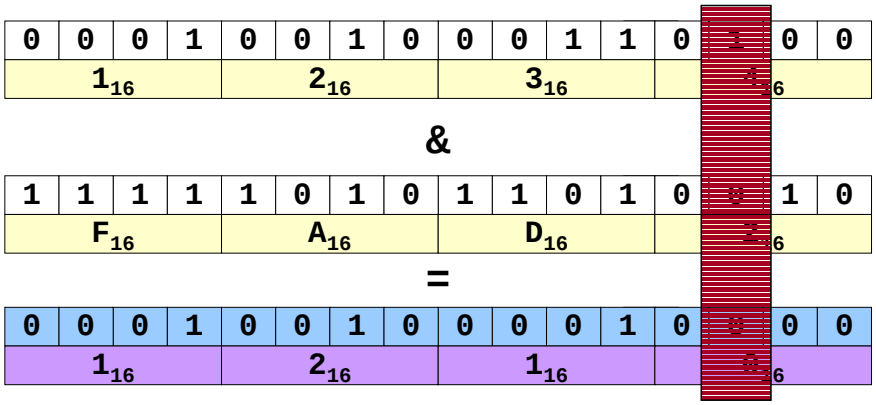
check bit to the left of rightmost 1

Bitwise AND

Logic operations are BITWISE:

- 1234_{16} & $FAD2_{16}$ = 1210_{16}

x	y	AND
0	0	0
0	1	0
1	0	0
1	1	1

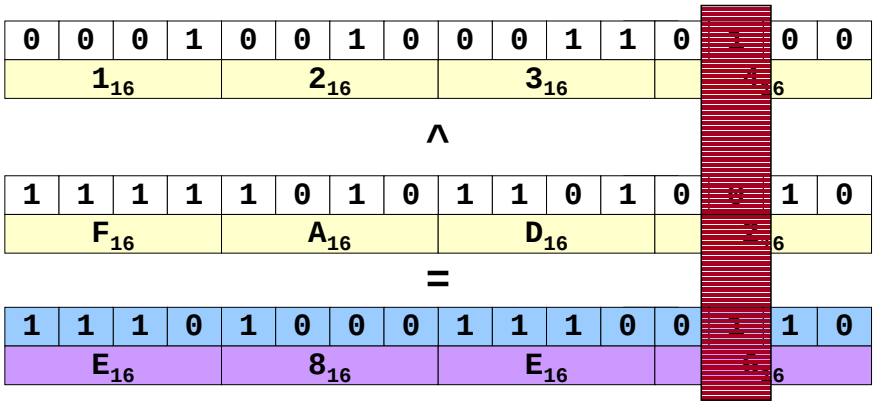


Bitwise XOR

Logic operations are BITWISE:

- 1234_{16} ^ $FAD2_{16}$ = $E8E6_{16}$

x	y	XOR
0	0	0
0	1	1
1	0	1
1	1	0



Dragon in XTOY

Impress your Yale friends with these 10 lines of code!

- Connect stdout to turtle.
  - L if zero
  - R if nonzero
- Keeps going until turtle runs out of energy.

dragon.toy

```
10: 7101   R[1] <- 0001
11: 7201   R[2] <- 0001          k

12: 2321   R[3] <- R[2] - R[1]   k - 1
13: 4423   R[4] <- R[2] ^ R[3]   k ^ (k-1)
14: 1541   R[5] <- R[4] + R[1]   (k^(k-1))+1
15: 3552   R[5] <- R[5] & R[2]   k & ((k^(k-1))+1)
16: 95FF   write R[5]
17: 1221   R[2] <- R[2] + R[1]
18: C012   goto 12
19: 0000   halt
```


Function Call: A Failed Attempt

Goal: x × y × z.

- Need two multiplications: x × y, (x × y) × z.

A failed attempt:

- Write multiply loop at 30-36.
- Calling program agrees to store arguments in registers A and B.
- Function agrees to leave result in register C.
- Call function with jump absolute to 30.
- Return from function with jump absolute.

Reason for failure.

function?

```
10: 8AFF
11: 8BFF
12: C0
13: 1AC0
14: 8BFF
15: C0
16: 9CFF
17: 0000

30: 7101
31: CA36
32: 1CCB
33: 2AA1
34: C031
35: C0
```


Multiplication Function

Calling convention.

- Jump to line 30.
- Store a and b in registers A and B.
- Return address in register F.
- Put result c = a × b in register C.
- Register 1 is scratch.
- Overwrites registers A and B.

function

```
10: 8AFF
11: 8BFF
12: FF
13: 1AC0
14: 8BFF
15: FF
16: 9CFF
17: 0000

30: 7C00
31: 7101
32: CA36
33: 1CCB
34: 2AA1
35: C032
36: E
```

function.toy

```
30: 7C00   R[C] ← 00
31: 7101   R[1] ← 01
32: CA36   if (R[A] == 0) goto 36
33: 1CCB   R[C] += R[B]
34: 2AA1   R[A]--
35: C032   goto 32
36: EF00   pc ← R[F]
```

return

opcode E
jump register

Multiplication Function Call

Client program to compute x × y × z.

- Read x, y, z from standard input.
- Note: PC is incremented before instruction is executed.
  - value stored in register F is correct return address

function.toy (cont)

```
10: 82FF   read R[2]             x
11: 83FF   read R[3]             y
12: 84FF   read R[4]             z

13: 1A20   R[A] ← R[2]           x
14: 1B30   R[B] ← R[3]           y
15: FF30   R[F] ← pc; goto 30    x * y

16: 1AC0   R[A] ← R[C]           (x * y)
17: 1B40   R[B] ← R[4]           z
18: FF30   R[F] ← pc; goto 30    (x * y) * z

19: 9CFF   write R[C]
1A: 0000   halt
```

opcode F
jump and link

R[F] ← 16

Function Call: One Solution

Contract between calling program and function:

- Calling program stores function parameters in specific registers.
- Calling program stores return address in a specific register.
  - jump-and-link
- Calling program sets PC to address of function.
- Function stores return value in specific register.
- Function sets PC to return address when finished.
  - jump register

What if you want a function to call another function?

An Efficient Multiplication Algorithm

Inefficient multiply.

- Load in integers a and b, and store c = a × b.
- Brute-force algorithm:
  - initialize c = 0
  - add b to c, a times

"Grade-school" multiplication.

Decimal

```
  1 2 3 4
* 1 5 1 2
-------
  2 4 6 8
1 2 3 4
6 1 7 0
1 2 3 4
-------
0 1 8 6 5 8 0 8
```

Binary

```
  1 0 1 1
* 1 1 0 1
-------
  1 0 1 1
0 0 0 0
1 0 1 1
1 0 1 1
-------
1 0 0 0 1 1 1 1
```


Binary Multiplication

Grade school binary multiplication algorithm to compute c = a × b.

- Initialize c = 0.
- Loop over i bits of b.
  - if $b_i = 0$, do nothing
  - if $b_i = 1$, shift a left i bits and add to c

b_i = i^{th} bit of b

```
  1 0 1 1
* 1 1 0 1
-------
  1 0 1 1
0 0 0 0
1 0 1 1
1 0 1 1
-------
1 0 0 0 1 1 1 1
```

Implement with built-in TOY shift instructions.

multiply-fast.c

```
int c = 0;
for (i = 15; i >= 0; i--) {
   if ((b >> i) & 1)
      c += (a << i);
```


Shift Left

Shift left. (opcode 5)

- Move bits to the left, padding with zeros as needed.
- 1234_{16} << 7_{16} = 1600_{16}

discard

0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0
1_{16}				2_{16}				3_{16}				4_{16}			

<< 7 =

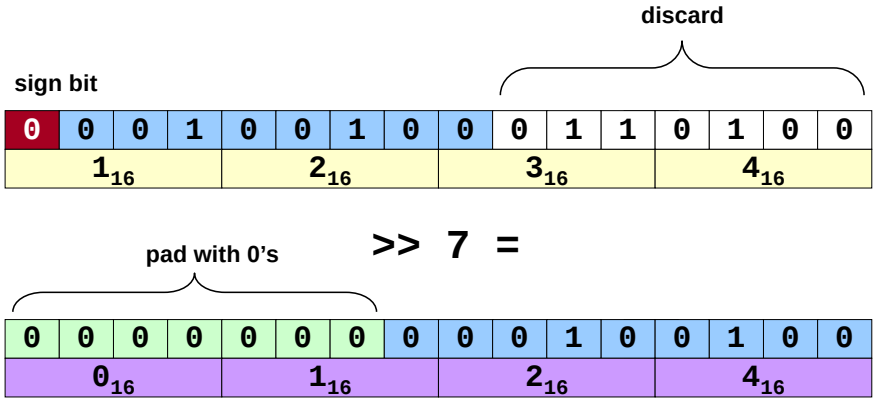
pad with 0's

0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0
1_{16}				6_{16}				0_{16}				0_{16}			

Shift Right

Shift left. (opcode 6)

- Move bits to the right, padding with sign bit as needed.
- 1234_{16} >> 7_{16} = 0124_{16}



Fast Multiplication Function

multiply-fast.toy

```
// Input:  R[A] contains a, R[B] contains b
// Output: R[C] contains a * b

30: 7101   R[1] <- 0001
31: 7210   R[2] <- 0010              i = 16
32: 7C00   R[C] <- 0000              result

33: C23B   if (R[2] == 0) goto 3B    while (i > 0) {
34: 2221   R[2]--                       i--
35: 53A2   R[3] ← R[A] << R[2]          a left shifted i
36: 64B2   R[4] ← R[B] >> R[2]
37: 3441   R[4] ← R[4] & R[1]           b_i = ith bit of b

38: C43A   if (R[4] == 0) goto 3A
39: 1CC3   R[C] += R[3]                 add to result

3A: C033   goto 33                   }
3B: EF00   goto R[F]                 return
```

16-bit integers

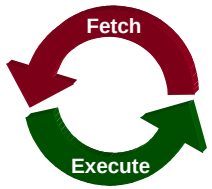
How To Build a TOY Machine

Implement fetch-execute cycle in hardware.

- See Lecture A4-A6.

X-TOY Machine

```
DO {
   fetch instruction
   execute instruction
} UNTIL (halt signal)
```



Implement fetch-execute cycle in software.

- Write a program to "simulate" the behavior of the TOY machine.
- TOY simulator in Java.
- TOY simulator in C.
- TOY simulator in TOY! (It has been done.)

TOY Simulator in C

TOY SIMULATOR: toy.c

```
int main(int argc, char *argv[]) {
   short R[16] = {0};         // registers
   short mem[256] = {0};      // memory
   unsigned char pc = 0x10;   // pc
   unsigned int inst, addr;
   int i, op, d, s, t;

   FILE *file;
   file = fopen(argv[1], "r");
   if (file == NULL) {
      printf("Error opening %s\n", file);
      exit(EXIT_FAILURE);
   }

   while (fscanf(file, "%2hX%4hX", &i, &inst) !=  EOF)
      mem[i] = inst;

   . . .
```

short = 16 bit 2's complement integer (on arizona)

open file for reading

Unix

```
% gcc toy.c
% a.out dragon.toy
```

read TOY code from file

X-TOY Simulator

xtoy.c: parse instruction

fetch and increment

s = bits 4-7

addr = bits 0-7

execute

register 0 always 0

```
do {
   inst = mem[pc++];
   op   = (inst >> 12) &  15;
   d    = (inst >>  8) &  15;
   s    = (inst >>  4) &  15;
   t    = (inst >>  0) &  15;
   addr = (inst >>  0) & 255;

   // see next slide
   . . .

   R[0] = 0;
} while (op != 0);

return 0;
}
```


Shifting and Masking

Extract destination register.

- Given 16 bit integer in C, isolate destination register (bits 8-11).
- Use bit operations in C.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
inst = 3604_{16}	0	0	1	1	0	1	1	0	0	0	0	0	0	1	0	0
inst >> 8	0	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0
15_{10}	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
(inst >> 8) & 15 = 6	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

TOY Simulator

toy.c: execute instruction

halt

bitwise AND

load address

load indirect

jump register

```
switch (op) {
   case  0:                              break;
   case  1: R[d] = R[s] +  R[t];         break;
   case  2: R[d] = R[s] -  R[t];         break;
   case  3: R[d] = R[s] &  R[t];         break;
   case  4: R[d] = R[s] ^  R[t];         break;
   case  5: R[d] = R[s] << R[t];         break;
   case  6: R[d] = R[s] >> R[t];         break;
   case  7: R[d] = addr;                 break;
   case  8: R[d] = mem[addr];            break;
   case  9: mem[addr] = R[d];            break;
   case 10: R[d] = mem[R[t]];            break;
   case 11: mem[R[t]] = R[d];            break;
   case 12: if (R[d] == 0) pc = addr;    break;
   case 13: if (R[d] >  0) pc = addr;    break;
   case 14: pc = R[d];                   break;
   case 15: R[d] = pc; pc = addr;        break;
}
```


TOY Simulator

Missing details for stdin, stdout.

- Stdin: insert following code before execute switch statement.
  - if address is FF and opcode is load or load indirect

```
if ((addr == 255 && op == 8) || (R[t] == 255 && op == 10))
   scanf("%4hX", &mem[255]);
```

- Stdout: insert following code after execute switch statement.
  - if address is FF and opcode is store or store indirect

```
if ((addr == 255 && op == 9) || (R[t] == 255 && op == 11))
       printf("%04hX\n", mem[255]);
```

Simulation

Consequences of simulation.

- Test out new machine (or microprocessor) using simulator.
  - cheaper and faster than building actual machine
- Easy to add new functionality to simulator.
  - trace, single-step, breakpoint debugging
  - simulator more useful than TOY itself
- Reuse software for old machines.

Ancient programs still running on modern computers.

- Ticketron.
- Lode Runner on Apple IIe.

Apple IIe Simulator

Basic Characteristics of X-TOY Machine

TOY is a general-purpose computer.

- Sufficient power to perform any computation.
- Limited only by amount of memory (and time).

Stored-program computer. (von Neumann memo, 1944)

- Data and instructions encoded in binary.
- Data and instructions stored in SAME memory.
- Can change program (control) without rewiring.
  - immediate applications
  - profound implications
- EDSAC (Wilkes 1949).
  - first stored-program computer
- Outgrowth of Turing's work.

All modern computers are general-purpose computers and have same (von Neumann) architecture.

Lecture A2: Supplemental Notes

TOY Cheat Sheet

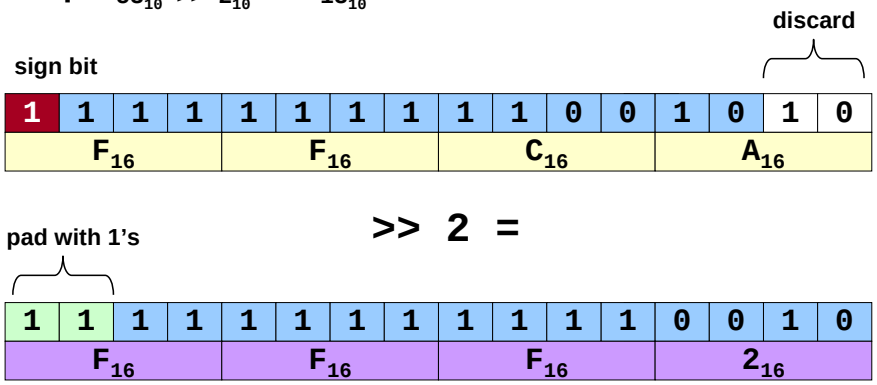
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Format 1	opcode				dest d				source s				source t			
Format 2	opcode				dest d				addr							

#	Operation	Fmt	Pseudocode
0:	halt	1	exit(0)
1:	add	1	R[d] ← R[s] + R[t]
2:	subtract	1	R[d] ← R[s] - R[t]
3:	and	1	R[d] ← R[s] & R[t]
4:	xor	1	R[d] ← R[s] ^ R[t]
5:	shift left	1	R[d] ← R[s] << R[t]
6:	shift right	1	R[d] ← R[s] >> R[t]
7:	load addr	2	R[d] ← addr
8:	load	2	R[d] ← mem[addr]
9:	store	2	mem[addr] ← R[d]
A:	load indirect	1	R[d] ← mem[R[t]]
B:	store indirect	1	mem[R[t]] ← R[d]
C:	branch zero	2	if (R[d] == 0) pc ← addr
D:	branch positive	2	if (R[d] > 0) pc ← addr
E:	jump register	2	pc ← R[d]
F:	jump and link	2	R[d] ← pc; pc ← addr

Shift Right

Shift right. (opcode 6)

- Move bits to the right, padding with sign bit as needed.
- $FFCA_{16}$ >> 2_{16} = $FFF2_{16}$
- -53_{10} >> 2_{10} = -13_{10}



Other Logical Operations

Any logical operation can be implemented with AND and XOR.

- See Boolean circuit lecture.

Build OR from AND and XOR.

- (x & y) ^ (x ^ y)

		a & b	a ^ b		
x	y	x & y	x ^ y	a ^ b	OR
0	0	0	0	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	0	1	1

Build NOT from XOR.

- 1 ^ x = x'
- FFFF ^ x = x' (bitwise NOT)

x	1	x ^ 1	NOT
0	1	1	1
1	1	0	0