PRE06.1. 06 Text and Videos

Videos

Video credits: Geoffrey Herman, Text credits: Geoffrey Herman

The Big Picture

We have now taught you about each of the modules that we can use to build an entire working processor. In other words, we can combine registers, multiplexers, arithmetic logic units, and finite state machines to create a fully functioning processor. This preflight and GA are here as a checkpoint/review to help you make sure you understand how these components work together before we start building a computer processor next class.

We will show you an example of an "Application Specific Integrated Circuit" (ASIC): a multiplication circuit. These types of circuits are used when you need to create a circuit where speed or low power are the most important goals and flexible programmability is not important. We use the FSM to "program" (i.e., control) the behavior and the other circuit components (i.e., the datapath). While we are "programming" this circuit with a FSM, we cannot write software for this ASIC.

The binary multiplication circuit is just an extended example to illustrate the relationship between the core concepts of datapaths and control. Binary multiplication and the multiplication circuit are not integral to the class. Our hope/goal is that you will be able to reason about complex circuits and their timing.

Binary Multiplication

Binary multiplication is performed with the same algorithm as decimal multiplication. To multiply A x B, start with the least-significant digit/bit of B and multiply that digit/bit from B with all A. Then repeat that process, but changing the bit of B and moving the product of the multiplication to the next most-significant bit position. After doing all the smaller multiplications, add the products together to get the final product (P).

Decimal Multiplication

```
    1234 (A)
x   1021 (B)
    1234
   2468
  0000
+ 1234
  1259914
```

Binary Multiplication

```
    1101 (A=13)
x   1010 (B=10)
    0000
   1101
  0000
+ 1101
  10000010 (P=130)
```

A bit shifting and adding algorithm for binary multiplication

The binary multiplication algorithm can be reconceived as a series of bit shifts and additions. Each iteration of the algorithm essentially shifts A to the left by 1 bit. Likewise, if we focus on extracting only the least-significant bit of B, we can get the same behavior as the above algorithm by shifting B to the right by 1 bit every iteration. Notes about the algorithm:

1. Initialize P^+=0. If A and B are both originally N bits (e.g., 4), then the product P can be up to 2N bits wide (e.g., 8), so P must be 8 bits wide.
2. Since we will shift A to the left each iteration, we need to zero extend it to 2N bits so that we don't lose the most significant bits when we bit shift (we want to lose the least-significant bits of B).
3. If B == 0, stop
4. Bit-mask B to see only its least-significant bit, and then multiply the masked B with A. Add that partial product to P.
5. Shift A to the left by one bit, Shift B to the right by one bit, then return to step 3.

Examples

4-bit Binary Shift-Add Multiplication

```
    1101 (A=13)
x   1010 (B=10)
```

Initialize P = 0 (in 8 bits), zero-extend A to 8 bits

```
 00001101
x    1010
```

Mask B to see only the least-significant bit

```
 00001101
x       0
 00000000
```

Add product to P => P = 0 + 0

Mask B to see only the least-significant bit

```
 00001101
x       0
 00000000
```

Add product to P => P = 0 + 0

Shift A to the left (`A <<= 1`)
Shift B to the right (`B >>= 1`)

```
 00011010
x    0101
```

Mask B to see only the least significant bit

```
 00011010
x       1
 00011010
```

Add product to P => P = 0 + 26

Shift A to the left (`A <<= 1`)
Shift B to the right (`B >>= 1`)

```
 00110100
x    0010
```

5-bit Binary Shift-Add Multiplication

```
    11110 (A=30)
x   01111 (B=15)
```

Initialize P = 0 (in 10 bits), zero-extend A to 10 bits

```
 0000011110
x    01111
```

Mask B to see only the least-significant bit

```
 0000011110
x        1
 0000011110
```

Add product to P => P = 0 + 30

Mask B to see only the least-significant bit

```
 0000011110
x        1
 0000011110
```

Add product to P => P = 0 + 30

Shift A to the left (`A <<= 1`)
Shift B to the right (`B >>= 1`)

```
 0000111100
x    00111
```

Mask B to see only the least significant bit

```
 0000111100
x        1
 0000111100
```

Add product to P => P = 30 + 60

Shift A to the left (`A <<= 1`)
Shift B to the right (`B >>= 1`)

```
 0001111000
x    00011
```

PRE 06

Assessment overview

Total points: 0/50

Score: 0%

Question PRE06.1

Value: 1

Total points: — /1

Auto-graded question

Previous question Next question

Personal Notes

No attached notes

Attach a file

Add text note

Mask B to see only the least significant bit

```
 00110100
x        0
 00000000
```

Add product to P => P = 26 + 0

Shift A to the left (`A <<= 1`)
Shift B to the right (`B >>= 1`)

```
 01101000
x    0001
```

Mask B to see only the least significant bit

```
 01101000
x        1
 01101000
```

Add product to P => P = 26 + 104

Shift A to the left (`A <<= 1`)
Shift B to the right (`B >>= 1`)

```
 11010000
x    0000
```

B is 0, so we can stop

P = 130

Mask B to see only the least significant bit

```
 0001111000
x         1
 0001111000
```

Add product to P => P = 90 + 120

Shift A to the left (`A <<= 1`)
Shift B to the right (`B >>= 1`)

```
 0011110000
x    00001
```

Mask B to see only the least significant bit

```
 0011110000
x         1
 0011110000
```

Add product to P => P = 210 + 240

Shift A to the left (`A <<= 1`)
Shift B to the right (`B >>= 1`)

```
 0111100000
x    00000
```

B is 0, so we can stop

P = 450

Multiplication Algorithm as Code

We can represent the multiplication algorithm with the example C code

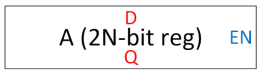
```c
// Example C code for the multiplication algorithm
// short are 16 bits
// int are 32 bits

int shift_add_multiply(unsigned short X, unsigned short Y) {
   unsigned int A = (unsigned int) X;  // zero-extend A to 2*16 bits
   unsigned short B = Y;               // 16 bit B
   unsigned int P = 0;                 // 32 bit product P

   while (B != 0) {        // Stop when B == 0
      if ((B & 1) == 1) {  // Mask B to see only the least-significant bit
         P = P + A;        // Add product to P
                           // P+ = P + A, the state of P after the clock edge will become P + A from before t
      }
      B = B >> 1;          // Shift B to the right; B+ = B >> 1
      A = A << 1;          // Shift A to the left; A+ = A << 1
   }
   return P;
}
```

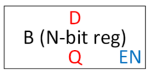
Multiplication Circuit and FSM

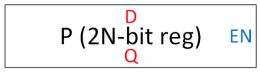
The multiplication ASIC circuit (below) will consist of three parts: state storage, state manipulation, and control

State storage

The algorithm stores the value (or data) of 3 variables: A (2N bits), B (N bits), and P (2N bits). We use one register to store each variable. These registers have an EN to control when we write to them. We may not want to write to all of them at the same time, so they each get their own "load" signal, Load A (LA), Load B (LB), and Load P (LP).







State manipulations

Let's consider how each state variable can be manipulated individually.

> Reminder: A^+ means the value is stored AFTER the next positive clock edge. For example, $P^+ = P + A$ means that the sum of the current values of P and A will become the new value stored in P after the next positive clock edge.

State manipulations of A

We perform only two potential state manipulations to A (i.e., assign a new value to A).

1. Store a new, zero-extended multicand in A. The zero-extend operation is illustrated by appending `N'b0` to the left of a bus shown in the figure below. `A+ = { N'b0, X }`
2. Shift A to the left: `A+ = A << 1;`

> `{ }` means bundle the wires in-between into a new bus. Wires on the left are in the more significant bit positions. For example, if `X==0x4`, `{ N'b0, X } == 0x04` not `0x40`.

Since we have two options for the next-state of A, we can use a multiplexer to choose between them, using a control signal

Select A SA.

State manipulations of B

We perform only two potential state manipulations to B (i.e., assign a new value to B).

1. Store a new multicand in B: `B+ = Y`
2. Shift B to the right: `B+ = B >> 1;`

Since we have two options for the next-state of B, we can use a multiplexer to choose between them, using a control signal Select B SB.

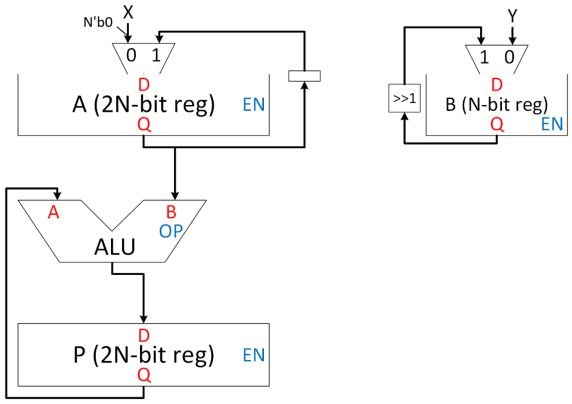
State manipulations of P

We perform only two potential state manipulations to P (i.e., assign a new value to P).

1. Initialize P to 0: `P+ = 0;`
2. Add the current value of A to P: `P+ = P + A;`

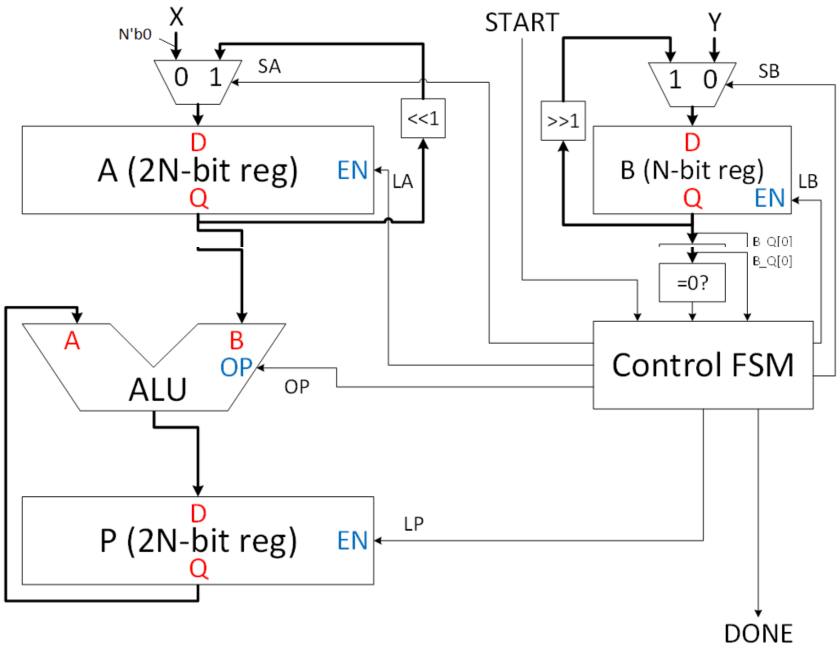
Since we have two options for the next-state of P, we can use a multiplexer to choose between them. Since we also need an adder, we hide this multiplexer inside an ALU that outputs 0 when a control signal OP==0 and outputs addition when OP==1.

Side note 1: The ALU module has its own inputs A and B. The A and B inside the ALU are NOT the same as the values stored in registers A and B. Every module can use whatever variable names it wants to internally. Similarly, registers A, B, and P all have their own D inputs, but each D input is a different instance of that variable in a different instance of the register module.



Control

We need to make three major decisions about the behavior of our circuit: 1) when to wait or start a new calculation (i.e., our function is not called or called), 2) when to continue or stop a current calculation (i.e., `B != 0`), and 3) when to add A to P or not based on the least-significant bit of B (`(B & 1) == 1`). Our control FSM therefore needs three pieces of information to create the program that controls our circuit: When to START (S), whether B is 0 (Zero (Z)) and the least-significant bit of B (B_Q[0] (B). These three signals (SZB) are the inputs to our control FSM. The outputs of our control FSM are all of the control signals we identified earlier: LA, LB, SA, SB, and OP. We also need one more control signal DONE (D) to indicate that our computation is complete (this is what the `return` statement does in the C code.



You will explore how to determine the states and state transitions of the FSM during class.

Datapath vs. Control

The circuit components described in State Storage and State Manipulations - registers, shifters, MUXes, and ALU - are the datapath of the ASIC. The FSM is the control part of the ASIC.

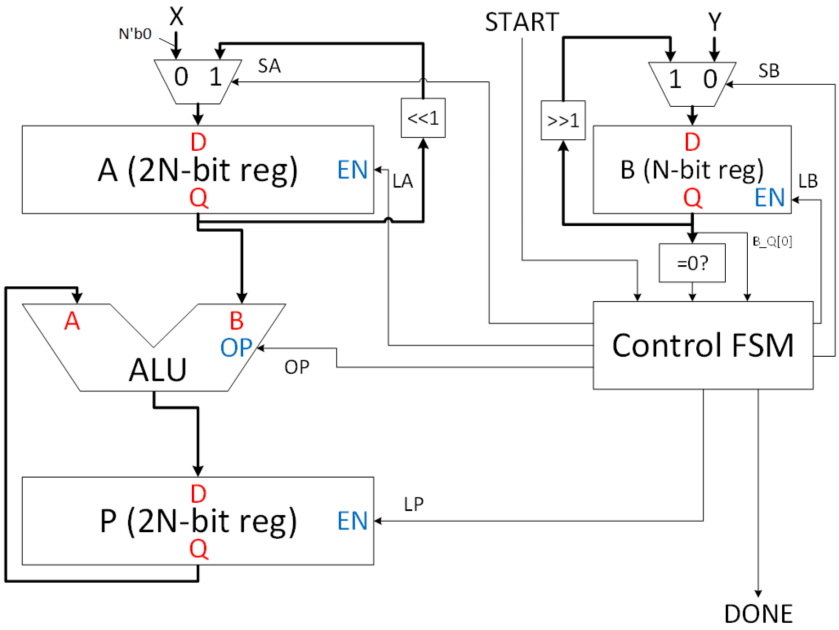
Datapath Examples

In these examples, Register A and Register P are 8-bits wide. Register B is 4-bits wide. When OP is 0, ALU out = 0. When OP is 1, ALU out = A + P. X is zero-extended from 4 to 8-bits (That's what the `N'b0` does).

> Note: In the tables below, the number of hex digits must match the number of bits in the data (i.e., A and P have 2 hex digits because they are 8 bits wide)

Assuming that the datapath has the input data, state values stored in the registers, and control signals shown below, what state will be stored in the registers after the next positive clock edge? Note: these examples may not reflect what you would actually want to do to implement multiplication, they are merely illustrative of what the control signals make happen.

the current values of P and A will become the new value stored in P after the next positive clock edge.



Data inputs

Input Name	Value
Input X	0x0
Input Y	0xb

Register Name	State
Reg A	0x5d
Reg B	0xa
Reg P	0x24

Control signals

Control Signal	Value
Load A (LA)	1
Shift A (SA)	1
Load B (LB)	1
Shift B (SB)	0
Load P (LP)	1
ALU operation (OP)	1

- Because LA==1, the value of A will change after the next clock edge. Because SA==1, the MUX chooses the value from the left-shift by 1 unit to send to register A.
  A^+ = A << 1 = 0xba
- Because LB==1, the value of B will change after the next clock edge. Because SB==0, the MUX chooses the value Y to send to register B.
  B^+ = Y = 0xb
- Because LP==1, the value of P will change after the next clock edge. Because OP==1, the ALU will send the sum from an addition to register P.
  P^+ = P + A = 0x24+0x5d=0x81

Note: All assignments to A, B, and P happen in parallel (i.e., simultaneously at the clock edge). This is why we use the original value of A (0x5d) in our addition rather than the next-state value of A^+ (0xba)

Data inputs

Input Name	Value
Input X	0xe
Input Y	0xe

Register Name	State
Reg A	0x64
Reg B	0x5
Reg P	0x67

Control signals

Control Signal	Value
Load A (LA)	1
Shift A (SA)	0
Load B (LB)	0
Shift B (SB)	1
Load P (LP)	1
ALU operation (OP)	1

- Because LA==1, the value of A will change after the next clock edge. Because SA==0, the MUX chooses the zero-extended value of X to send to register A.
  A^+ = { 4'b0, X } = 0x0e
- Because LB==0, the value of B will not change after the next clock edge.
  B^+ = B = 0x5
- Because LP==1, the value of P will change after the next clock edge. Because OP==1, the ALU will send the sum from an addition to register P.
  P^+ = P + A = 0x67+0x64=0xcb

Note: All assignments to A, B, and P happen in parallel (i.e., simultaneously at the clock edge). This is why we use the original value of A (0x64) in our addition rather than the next-state value of A^+ (0x0e)

Mark as read