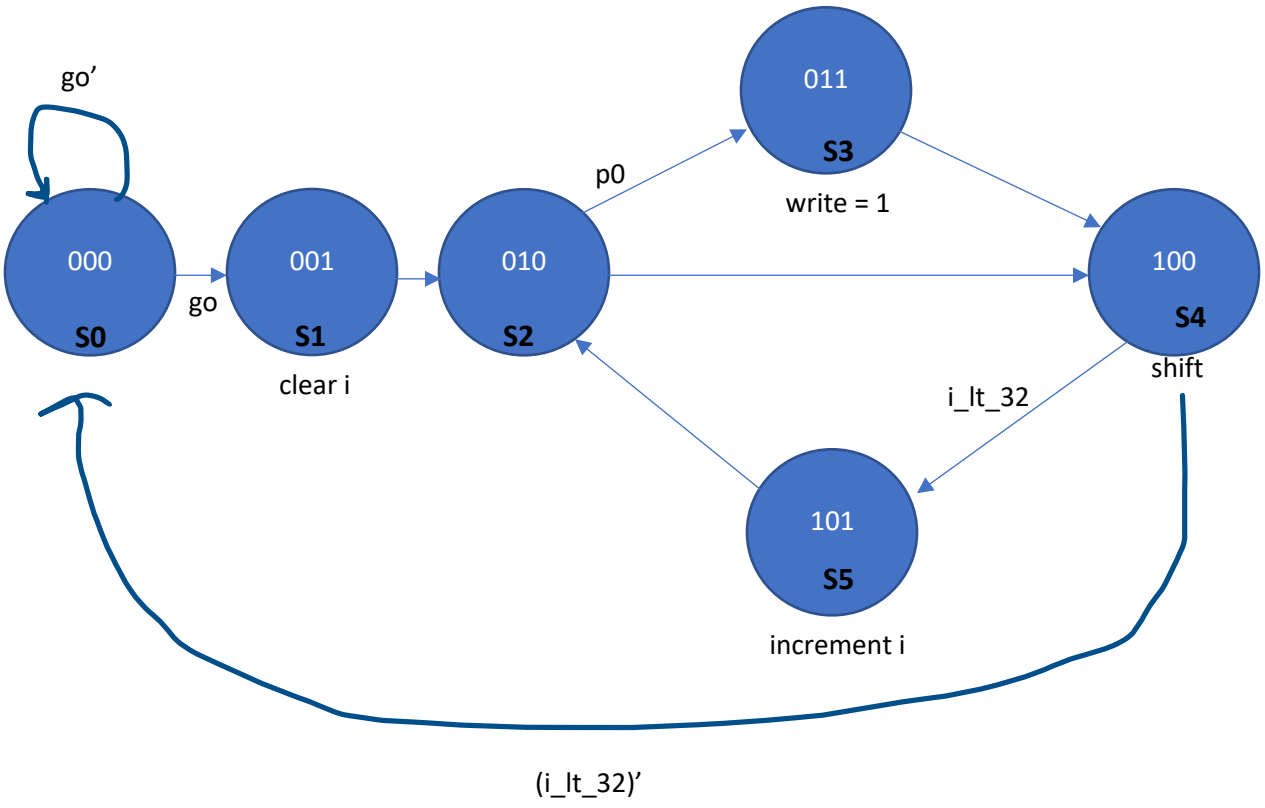
COMPUTER ORGANIZATION (CSE 331) – 2ND HOMEWORK

MERT GÜRŞİMŞİR – 1901042646

FSM for The Control Unit of the Multiplier



Here, I tried to implement ASM given in the homework PDF as FSM so that I can write the code looking from this machine. Then I have implemented this FSM in the control.v file.

I couldn't fully manage to implement multiplication. So some parts are missing.

control.v

Here, I tried to implement the ASM by looking at the code that we have written in the problem session.

First we determine the present state and later the next state by using present state.

Inputs are deciding what we are going to do. For example if i is (which is counter) less than 32, we continue with S5, the next state, but if not we continue from beginning.

datapath.v

Here, we implement the decisions.

If write is 1, I added multiplicand to the left half of the product.

I shifted the product right by 1, incremented the i, control if i is less than 32 or not and clear the i for the restart.

alu32.v

We have 8 operations and 3 bits that represent which operation is going to be done as inputs, and Result as output.

We have 8 different modules that are called from alu which are:

- _32_bit_adder
- _32_bit_subtractor
- _32_bit_set_less_than
- _32_bits_xor
- _32_bits_and
- _32_bits_or
- _32_bits_nor
- _8x1_mux

Decimal results for ADD (000), SUB (001), SLT (110):

```
# ALU Results: Result =          50, A =          16, B =          34, ALUop = 000
# ALU Results: Result =         200, A =         100, B =         100, ALUop = 000
# ALU Results: Result =          53, A =         100, B =          47, ALUop = 001
# ALU Results: Result =         102, A =         105, B =           3, ALUop = 001
# ALU Results: Result = 4294967295, A = 4294967295, B =           0, ALUop = 011
# ALU Results: Result = 2868546811, A = 2883523124, B =    19242703, ALUop = 011
# ALU Results: Result =           0, A = 4294967295, B =           0, ALUop = 100
# ALU Results: Result =    17109508, A = 2883523124, B =    19242703, ALUop = 100
# ALU Results: Result = 4294967295, A = 4294967295, B =           0, ALUop = 101
# ALU Results: Result = 2885656319, A = 2883523124, B =    19242703, ALUop = 101
# ALU Results: Result =           0, A =         100, B =          47, ALUop = 110
# ALU Results: Result =           1, A =         100, B =         147, ALUop = 110
# ALU Results: Result =           0, A = 4294967295, B =           0, ALUop = 111
# ALU Results: Result = 1409310976, A = 2883523124, B =    19242703, ALUop = 111
```

Binary results for XOR, (011), AND (100), OR (101), NOR (111):

```
# ALU Results: Result = 00000000000000000000000000110010, A = 00000000000000000000000000010000, B = 00000000000000000000000000100010, ALUop = 000
# ALU Results: Result = 00000000000000000000000011001000, A = 00000000000000000000000001100100, B = 00000000000000000000000001100100, ALUop = 000
# ALU Results: Result = 00000000000000000000000000110101, A = 00000000000000000000000001100100, B = 00000000000000000000000000101111, ALUop = 001
# ALU Results: Result = 00000000000000000000000011001110, A = 00000000000000000000000011001001, B = 00000000000000000000000000000111, ALUop = 001
# ALU Results: Result = 11111111111111111111111111111111, A = 11111111111111111111111111111111, B = 00000000000000000000000000000000, ALUop = 011
# ALU Results: Result = 10101010111110101000110011111011, A = 10101011110111111000100100110100, B = 00000001001001011001111011001111, ALUop = 011
# ALU Results: Result = 00000000000000000000000000000000, A = 11111111111111111111111111111111, B = 00000000000000000000000000000000, ALUop = 100
# ALU Results: Result = 00000001000001010001001000000100, A = 10101011110111110001001000110100, B = 00000001001001011001111011001111, ALUop = 100
# ALU Results: Result = 11111111111111111111111111111111, A = 11111111111111111111111111111111, B = 00000000000000000000000000000000, ALUop = 101
# ALU Results: Result = 10101011111111111001111011111111, A = 10101011110111110001001000110100, B = 00000001001001011001111011001111, ALUop = 101
# ALU Results: Result = 00000000000000000000000000000000, A = 00000000000000000000000001100100, B = 00000000000000000000000000101111, ALUop = 110
# ALU Results: Result = 00000000000000000000000000000001, A = 00000000000000000000000001100100, B = 00000000000000000000000000010011, ALUop = 110
# ALU Results: Result = 00000000000000000000000000000000, A = 11111111111111111111111111111111, B = 00000000000000000000000000000000, ALUop = 111
# ALU Results: Result = 01010100000000000110000100000000, A = 10101011110111110001001000110100, B = 00000001001001011001111011001111, ALUop = 111
```

_32_bit_adder.v

We have A, B as 32 bit number, CIn as carry in as inputs; S as result, COut as carry out, and C30 as carry out of 30th bit. We call full adder module for each bit.

```
# 32 Bit Adder Test: A =          16, B =          34, RESULT =          50, Cin = 0, Cout = 0
# 32 Bit Adder Test: A =         165, B =         128, RESULT =         293, Cin = 0, Cout = 0
# 32 Bit Adder Test: A =         100, B =         100, RESULT =         200, Cin = 0, Cout = 0
# 32 Bit Adder Test: A =           0, B =           0, RESULT =           1, Cin = 1, Cout = 0

# 32 Bit Adder Test: A = 4294967295, B =           1, RESULT =           0, Cin = 0, Cout = 1
```

full_adder.v

Simple full adder module that is created by 2 half adders and 1 or gate. Takes A, B, Cin as inputs and gives Cout, Sum as outputs.

```
# Full Adder Test: A = 0, B = 0, RESULT = 0, Cout = 0, Cin = 0
# Full Adder Test: A = 0, B = 1, RESULT = 1, Cout = 0, Cin = 0
# Full Adder Test: A = 1, B = 0, RESULT = 1, Cout = 0, Cin = 0
# Full Adder Test: A = 1, B = 0, RESULT = 0, Cout = 1, Cin = 1
```

half_adder.v

Simple half adder implementation using 1 xor and 1 and gates. Takes A, B as inputs and gives Sum, Carry as outputs.

```
# Half Adder Test: RESULT = 0, A = 0, B = 0, Cout = 0
# Half Adder Test: RESULT = 1, A = 0, B = 1, Cout = 0
# Half Adder Test: RESULT = 1, A = 1, B = 0, Cout = 0
# Half Adder Test: RESULT = 0, A = 1, B = 1, Cout = 1
```

_32_bit_subtractor.v

Here, first I get the not of the B (the second number) and than add A and this B's not by giving carry in as 1. With this way, I take negative of B (two's complement) and make the subtraction.

This module takes A, B as inputs and gives SUB (result), Cout, C30 as outputs.

```
# 32 Bit Subtractor Test: A =         34, B =         16, RESULT =         18
# 32 Bit Subtractor Test: A =        165, B =        128, RESULT =         37
# 32 Bit Subtractor Test: A =        100, B =         40, RESULT =         60
# 32 Bit Subtractor Test: A =          0, B =          0, RESULT =          0
```

_32_bit_set_less_than.v

I simply did subtraction for 2 numbers.

If result is negative and there is no overflow, simply take the last bit of the result (sign bit).
If result is negative and there is overflow, take the reverse of the last bit.

With this way, we can decide if a number is smaller than other one.

```
# 32 Bit Set-Less-Than Test: A =         34, B =         16, Set = 0
# 32 Bit Set-Less-Than Test: A =        165, B =        166, Set = 1
# 32 Bit Set-Less-Than Test: A =        100, B =         99, Set = 0
# 32 Bit Set-Less-Than Test: A =          0, B =          0, Set = 0
```

_32_bits_xor/and/or/nor.v

In this modules, I simply apply the operation on each bit. These modules takes numbers A, B and gives output Res.

```
# 32 Bit XOR Test: A = 10101011110011011110111100010001, B = 11111111111111111111111111111111, RESULT = 01010100001100100001000011101110
# 32 Bit XOR Test: A = 00000000101010100000000000000000, B = 00001100110100001011101100000000, RESULT = 00001100011110101011101100000000
# 32 Bit XOR Test: A = 11110000000000000000000000000001, B = 10100000000000001001100000000001, RESULT = 01010000000000001001100000000000
```

```
# 32 Bit AND Test: A = 10101011110011011110111100010001, B = 11111111111111111111111111111111, RESULT = 10101011110011011110111100010001
# 32 Bit AND Test: A = 00000000101010100000000000000000, B = 00001100110100001011101100000000, RESULT = 00000000100000000000000000000000
# 32 Bit AND Test: A = 11110000000000000000000000000001, B = 10100000000000001001100000000001, RESULT = 10100000000000000000000000000001
```

```
# 32 Bit OR Test: A = 10101011110011011110111100010001, B = 11111111111111111111111111111111, RESULT = 11111111111111111111111111111111
# 32 Bit OR Test: A = 00000000101010100000000000000000, B = 00001100110100001011101100000000, RESULT = 00001100111111010101110110000000
# 32 Bit OR Test: A = 11110000000000000000000000000001, B = 10100000000000001001100000000001, RESULT = 11110000000000001001100000000001
```

```
# 32 Bit NOR Test: A = 10101011110011011110111100010001, B = 11111111111111111111111111111111, RESULT = 00000000000000000000000000000000
# 32 Bit NOR Test: A = 00000000101010100000000000000000, B = 00001100110100001011101100000000, RESULT = 11110011000001010100010011111111
# 32 Bit NOR Test: A = 11110000000000000000000000000001, B = 10100000000000001001100000000001, RESULT = 00001111111111110110011111111110
```

_8x1_mux.v

Simple 8x1 mux implementation using 2x1 mux we have seen in Logic. These module uses seven 2x1 mux to implement a 8x1 mux. Modules chooses among seven 1-bit inputs by using input S and put result in RES.

```
# 8x1 MUX Test: RESULT = 1, I0 = 1, I1 = 0, I2 = 0, I3 = 0, I4= 0, I5 = 0, I6 = 0, I7 = 0, S = 000
# 8x1 MUX Test: RESULT = 0, I0 = 1, I1 = 0, I2 = 1, I3 = 1, I4= 1, I5 = 1, I6 = 1, I7 = 1, S = 001
# 8x1 MUX Test: RESULT = 0, I0 = 1, I1 = 1, I2 = 1, I3 = 1, I4= 1, I5 = 1, I6 = 1, I7 = 0, S = 111
```

_2x1_mux.v

Simple 2x1 mux implementation using not, and, or gates. Module chooses among two 1-bit inputs by using given S bit and put result in RES.

```
# 2x1 MUX Test: RESULT = 0, I0 = 0, I1 = 0, S = 1
# 2x1 MUX Test: RESULT = 1, I0 = 0, I1 = 1, S = 1
# 2x1 MUX Test: RESULT = 0, I0 = 1, I1 = 0, S = 1
# 2x1 MUX Test: RESULT = 1, I0 = 1, I1 = 0, S = 0
```