/// **LSU EE 3755 -- Spring 2002 -- Computer Organization**

//

/// **Verilog Notes 7 -- Integer Multiply and Divide**

// Time-stamp: <13 March 2002, 10:18:27 CST, koppel@sol>

/// **Contents**

//

// Binary Multiplication Algorithm

// Simple Multiplication Hardware

// Streamlined Multiplication Hardware

// Booth Recoding for Higher-Radix and Signed Multiplication

// Binary Division Algorithm

// Division Hardware (Simple and Streamlined)

// "Real" Arithmetic Circuits

/// **References**

//

// :P: Palnitkar, "Verilog HDL"

// :Q: [Qualis, "Verilog HDL Quick Reference Card Revision 1.0"](http://www.ece.lsu.edu/v/refcard.pdf)

// :H: [Hyde, "Handbook on Verilog HDL"](http://www.eg.bucknell.edu/~cs320/1995-fall/verilog-manual.html)

// :LRM: [IEEE, Verilog Language Reference Manual (Hawaii Section Numbering)](http://www-ee.eng.hawaii.edu/~msmith/ASICs/HTML/Verilog/Verilog.htm)

// :PH: Patterson & Hennessy, "Computer Organization & Design"

// :HP: Hennessy & Patterson, "Computer Architecture: A Quantitative Approach"

////////////////////////////////////////////////////////////////////////////////

/// **Binary Multiplication Algorithm**

// :PH: 4.6

/// **Long Hand Procedure Review**

//

// Multiply 5 times 12 in binary:

//

// 0101 Multiplicand

// 1100 Multiplier

// """"

// 0000

// 0000

// 0101

// 0101

// """""""

// 0111100 Product

////////////////////////////////////////////////////////////////////////////////

/// **Simple Multiplication Hardware**

// :PH: 4.6

/// **Simple Multipliers**

//

// Designed to show basic techniques, but are inefficient.

// Following sections show improved multipliers.

/// **Multipliers in this section:**

//

// Simple Unsigned Combinational Multiplier

// See :PH: Figure 4.25

// Multiplies using combinational logic only.

// Cost of n-bit multiplier: proportional to n^2.

// Speed of n-bit multiplier: time needed for n additions:

// Proportional to n.

//

// Simple Unsigned Multiplier

// Unlike combinational multiplier, uses a clock.

// Cost of n-bit multiplier:

// Proportional to n.

// Includes 5n bits of register storage.

// Speed of n-bit multiplier:

// Product in n clocks.

// Clock period includes 2n-bit add.

// Uses more hardware than streamlined multipliers in next section.

// :**Example**:

//

// Simple Unsigned Combinational Multiplier

// Based on :PH: Figure 4.25

//

// This is a straightforward coding of the longhand multiplication

// technique.

module **simple\_combinational\_mult**(product,multiplier,multiplicand); //P

input [15:0] multiplier, multiplicand;

output product;

reg [31:0] product;

integer i;

always @( multiplier or multiplicand )

begin

product = 0;

for(i=0; i<16; i=i+1)

if( multiplier[i] == 1'b1 ) product = product + ( multiplicand << i );

end

endmodule

// :**Example**:

//

// Simple Unsigned Multiplier

//

// Sequential multiplier. Handshaking signals are added to command

// multiplier to start and to indicate when finished. Registers are

// provided for shifted versions of the multiplier and multiplicand.

module **simple\_mult**(product,ready,multiplier,multiplicand,start,clk); //P

input [15:0] multiplier, multiplicand;

input start, clk;

output product;

output ready;

reg [31:0] product;

reg [15:0] multiplier\_copy;

reg [31:0] multiplicand\_copy;

reg [4:0] bit;

wire **ready** = !bit;

initial bit = 0;

always @( posedge clk )

if( ready && start ) begin

bit = 16;

product = 0;

multiplicand\_copy = { 16'd0, multiplicand };

multiplier\_copy = multiplier;

end else if( bit ) begin

if( multiplier\_copy[0] == 1'b1 ) product = product + multiplicand\_copy;

multiplier\_copy = multiplier\_copy >> 1;

multiplicand\_copy = multiplicand\_copy << 1;

bit = bit - 1;

end

endmodule

////////////////////////////////////////////////////////////////////////////////

/// **Streamlined Multiplication Hardware**

// :PH: 4.6

/// **Streamlined Multipliers**

//

// Uses less hardware than simple multiplier.

// Much slower than "real" multipliers.

/// **Multipliers in this section:**

//

// Streamlined Unsigned Multiplier

// See :PH: Figure 4.31

// Less register storage than simple multiplier.

// Cost of n-bit multiplier:

// Proportional to n.

// Includes 2n bits of register storage.

// Speed of n-bit multiplier: (Same as simple.)

// Product in n clocks.

// Clock period includes a 2n-bit add.

//

// Streamlined Signed Multiplier

// Like previous multiplier, but handles signed numbers.

// A better signed multiplier covered later (using Booth recoding).

// Cost of n-bit multiplier:

// Proportional to n.

// Includes 3n bits of register storage.

// Speed of n-bit multiplier:

// Product in n clocks.

// Clock period includes 2n-bit add.

// :**Example**:

//

// Streamlined Unsigned Multiplier

// Based on :PH: Figure 4.31

//

// Improvements:

//

// The product register also holds the multiplier.

// Only the product register is shifted.

// A register is not needed to hold the multiplicand.

// Wow, you save 2n bits! (To multiply n-bit numbers.)

module **streamlined\_mult**(product,ready,multiplier,multiplicand,start,clk);

input [15:0] multiplier, multiplicand;

input start, clk;

output product;

output ready;

reg [31:0] product;

reg [4:0] bit;

wire **ready** = !bit;

initial bit = 0;

always @( posedge clk )

if( ready && start ) begin

bit = 16;

product = { 16'd0, multiplier };

end else if( bit ) begin:A

reg lsb;

lsb = product[0];

product = product >> 1;

bit = bit - 1;

if( lsb ) product[31:15] = product[30:15] + multiplicand;

end

endmodule

// :**Example**:

//

// Streamlined Signed Multiplier

//

// Multiplies absolute value of multiplier and multiplicand.

// Negates product if multiplier and multiplicand differ in sign.

// Otherwise similar to streamlined unsigned multiplier.

module **streamlined\_signed\_mult**(product,ready,multiplier,multiplicand,

start,clk);

input [15:0] multiplier, multiplicand;

input start, clk;

output product;

output ready;

reg [31:0] product;

reg [15:0] abs\_multiplicand;

reg [4:0] bit;

wire **ready** = !bit;

initial bit = 0;

always @( posedge clk )

if( ready && start ) begin

bit = 16;

product = { 16'd0, multiplier[15] ? -multiplier : multiplier };

abs\_multiplicand = multiplicand[15] ? -multiplicand : multiplicand;

end else if( bit ) begin:A

reg lsb;

lsb = product[0];

product = product >> 1;

bit = bit - 1;

if( lsb ) product[31:15] = product[30:15] + abs\_multiplicand;

if( !bit && multiplicand[15] ^ multiplier[15] ) product = -product;

end

endmodule

////////////////////////////////////////////////////////////////////////////////

/// **Booth Recoding for Higher-Radix and Signed Multiplication**

// :PH: 4.6 Only describes Radix-2 Booth Recoding

/// **Basic Idea**

//

// Rather than repeatedly adding either 0 or 1 times multiplicand,

// repeatedly add 0, 1, -1, 2, -2, etc. times the multiplicand.

//

/// **Benefits**

//

// Performs signed multiplication without having to first compute the

// absolute value of the multiplier and multiplicand.

//

// Improved performance (when radix higher than 2).

/// **Multipliers Described Here**

//

// Ordinary Radix-4 Unsigned Multiplier

// Presented for pedagogical reasons, Booth multipliers better.

// Twice as fast as earlier multipliers.

// Uses more hardware than Booth multipliers below.

//

// Booth Recoding Radix-4 Multiplier

// Multiplies signed numbers.

// Twice as fast as earlier multipliers.

//

// Booth Recoding Radix-2 Multiplier

// Multiplies signed numbers.

// Uses about the same amount of hardware than earlier signed multiplier.

/// **Ordinary Radix-4 Multiplier Idea**

//

// Review of Radix-2 Multiplication

//

// Multiply 5 times 12. Radix-2 multiplication (the usual way).

//

// 0101 Multiplicand

// 1100 Multiplier

// """"

// 0000 0 x 0101

// 0000 0 x 0101

// 0101 1 x 0101

// 0101 1 x 0101

// """""""

// 0111100 Product

//

// Radix-4 Multiplication

// Let "a" denote the multiplicand and b denote the multiplier.

// Pre-compute 2a and 3a.

// Examine multiplier two bits at a time (rather than one bit at a time).

// Based on the value of those bits add 0, a, 2a, or 3a (shifted by

// the appropriate amount).

//

// Uses n/2 additions to multiply two n-bit numbers.

//

// Two Radix-4 Multiplication Examples

//

// Multiply 5 times 12. Radix-4 multiplication (the faster way).

//

// Precompute: 2a: 01010, 3a: 01111

//

// 0101 Multiplicand

// 1100 Multiplier

// """"

// 00000 00 x 0101

// 01111 11 x 0101

// """""""

// 0111100 Product

//

// Multiply 5 times 9. Radix-4 multiplication (the faster way).

//

// 0101 Multiplicand

// 1001 Multiplier

// """"

// 00101 01 x 0101

// 01010 10 x 0101

// """""""

// 0101101 Product

// Ordinary Radix-2^d Multiplier

//

// This is a generalization of the Radix-4 multiplier.

//

// Let "a" denote the multiplicand and b denote the multiplier.

// Pre-compute 2a, 3a, 4a, 5a, ..., (2^d-1)a

// Examine multiplier d bits at a time.

// Let the value of those bits be v.

// Add v shifted by the appropriate amount.

//

// Uses n/d additions to multiply two n-bit numbers.

// :**Example**:

//

// A Radix-4 multiplier. Takes unsigned numbers.

module **imult\_ord\_radix\_4**(prod,ready,multiplicand,multiplier,start,clk);

input [15:0] multiplicand, multiplier;

input start, clk;

output prod;

output ready;

reg [32:0] product;

wire [31:0] **prod** = product[31:0];

reg [4:0] bit;

wire **ready** = !bit;

reg [17:0] pp;

initial bit = 0;

wire [17:0] **multiplicand\_X\_1** = {1'b0,multiplicand};

wire [17:0] **multiplicand\_X\_2** = {multiplicand,1'b0};

wire [17:0] **multiplicand\_X\_3** = multiplicand\_X\_2 + multiplicand\_X\_1;

always @( posedge clk )

if( ready && start ) begin

bit = 8;

product = { 16'd0, multiplier };

end else if( bit ) begin

case ( {product[1:0]} )

2'd0: pp = {2'b0, product[31:16] };

2'd1: pp = {2'b0, product[31:16] } + multiplicand\_X\_1;

2'd2: pp = {2'b0, product[31:16] } + multiplicand\_X\_2;

2'd3: pp = {2'b0, product[31:16] } + multiplicand\_X\_3;

endcase

product = { pp, product[15:2] };

bit = bit - 1;

end

endmodule

/// **Booth Recoding**

//

// Goal: Reduce the number of pre-computed multiples of the multiplicand.

// Bonus: Handles signed numbers.

//

// Idea: Use something like carries.

//

// Why It's Called Recoding

// Numbers like 3 are recoded into 4-1, avoiding the need to deal with 3.

//

// Consider need to compute multiplicand\_X\_3 for radix-4 multiplier.

// Instead:

// Subtract multiplicand and add 4 times multiplicand.

// The subtraction is done in the "current" step.

// The addition is done in the "next" step.

// Multiply 5 times 3. Booth radix-4 multiplication.

//

// 0101 Multiplicand

// 0011 Multiplier

// """""""

// 1111011 -1 x 0101 Multiplier bits 11 -> -1 + 4 -> -1 now, +1 next step

// 00101 (00+1) x 0101 Multiplier bits 00 -> 0 (now) + 1 (from last step)

// """""""

// 0001111 Product

// The table below indicates:

//

// MB: Multiplier bits value.

// C: Carry bit (lostbit in modules) from previous step.

// x: Amount to multiply multiplicand by.

// c: Carry bit to next step.

//

// Radix-4 Booth Table

//

// MB C | x c

// """""""""""""""""

// 00 0 | 0 + 0 0

// 00 1 | 1 + 0 0

// 01 0 | 1 + 0 0

// 01 1 | 2 + 0 0

// 10 0 | -2 + 4 1

// 10 1 | -1 + 4 1

// 11 0 | -1 + 4 1

// 11 1 | 0 + 4 1

//

// For example above:

// Step 1: MB -> 11, C -> 0, lookup reveals x -> -1, c -> 1

// Step 2: MB -> 00, C -> 1, lookup reveals x -> 1, c -> 0

//

// Radix-2 Booth Table

//

// MB C | x c

// """""""""""""""""

// 0 0 | 0 + 0 0

// 0 1 | 1 + 0 0

// 1 0 | -1 + 2 1

// 1 1 | 0 + 2 1

// Note:

// The carry out is the same as the MSB of the multiplier bits examined.

// That's intentional, so that no special storage or computation is need

// for the carry.

//

// The carry out of the last step is not used.

//

// Construction of higher-radix tables is straightforward.

// :**Example**:

//

// A Booth radix-4 multiplier. This takes signed numbers.

module **imult\_Booth\_radix\_4**(prod,ready,multiplicand,multiplier,start,clk); // P

input [15:0] multiplicand, multiplier;

input start, clk;

output prod;

output ready;

reg [32:0] product;

wire [31:0] **prod** = product[31:0];

reg [4:0] bit;

wire **ready** = !bit;

reg lostbit;

initial bit = 0;

wire [16:0] **multsx** = {multiplicand[15],multiplicand};

always @( posedge clk )

if( ready && start ) begin

bit = 8;

product = { 17'd0, multiplier };

lostbit = 0;

end else if( bit ) begin:A

case ( {product[1:0],lostbit} )

3'b001: product[32:16] = product[32:16] + multsx;

3'b010: product[32:16] = product[32:16] + multsx;

3'b011: product[32:16] = product[32:16] + 2 \* multiplicand;

3'b100: product[32:16] = product[32:16] - 2 \* multiplicand;

3'b101: product[32:16] = product[32:16] - multsx;

3'b110: product[32:16] = product[32:16] - multsx;

endcase

lostbit = product[1];

product = { product[32], product[32], product[32:2] };

bit = bit - 1;

end

endmodule

// :**Example**:

//

// Radix-2 Booth multiplier. Unlike signed multiplier, does not

// require time or hardware for taking absolute value of operands.

module **imult\_Booth**(product,ready,multiplicand,multiplier,start,clk); // P

input [15:0] multiplicand, multiplier;

input start, clk;

output product;

output ready;

reg [31:0] product;

reg [4:0] bit;

wire **ready** = !bit;

reg lostbit;

initial bit = 0;

always @( posedge clk )

if( ready && start ) begin

bit = 16;

product = { 16'd0, multiplier };

lostbit = 0;

end else if( bit ) begin:A

case ( {product[0],lostbit} )

2'b01: product[31:16] = product[31:16] + multiplicand;

2'b10: product[31:16] = product[31:16] - multiplicand;

endcase

lostbit = product[0];

product = { product[31], product[31:1] };

bit = bit - 1;

end

endmodule

////////////////////////////////////////////////////////////////////////////////

/// **Binary Division Algorithm**

// :PH: 4.7

// Binary Version of Longhand Division Technique

//

// 11 divided by 3:

//

// 11 (1011) is dividend.

// 3 (0011) is divider.

//

// """"""""|

// 1011 |

// -0011 |

// """"""""| 0 Difference is negative: copy dividend and put 0 in quotient.

// 1011 |

// -0011 |

// """"""""| 00 Difference is negative: copy dividend and put 0 in quotient.

// 1011 |

// -0011 |

// """"""""| 001 Difference is positive: use difference and put 1 in quotient.

// 0101 |

// -0011 |

// """"""""| 0011 Difference is positive: use difference and put 1 in quotient.

// 10 |

//

// Quotient, 3 (0011); remainder 2 (10).

////////////////////////////////////////////////////////////////////////////////

/// **Division Hardware (Simple and Streamlined)**

// :PH: 4.7

/// **Division Hardware**

//

// Simple Divider

// See :PH: Figure 4.36

// A straightforward translation of binary division algorithm into hardware.

// Cost of n-bit divider:

// Proportional to n.

// Uses 5n bits of register storage.

// Speed of n-bit divider:

// Quotient in n clock cycles.

//

// Streamlined Divider

// See :PH: Figure 4.41

// Less storage needed.

// Cost of n-bit divider:

// Proportional to n.

// Uses 2n bits of register storage.

// Speed of n-bit divider:

// Quotient in n clock cycles.

// :**Example**:

//

// Simple divider.

// Based on :PH: Figure 4.36

module **simple\_divider**(quotient,remainder,ready,dividend,divider,start,clk);

input [15:0] dividend,divider;

input start, clk;

output quotient,remainder;

output ready;

// """"""""|

// 1011 | <---- dividend\_copy

// -0011 | <---- divider\_copy

// """"""""| 0 Difference is negative: copy dividend and put 0 in quotient.

// 1011 | <---- dividend\_copy

// -0011 | <---- divider\_copy

// """"""""| 00 Difference is negative: copy dividend and put 0 in quotient.

// 1011 | <---- dividend\_copy

// -0011 | <---- divider\_copy

// """"""""| 001 Difference is positive: use difference and put 1 in quotient.

// quotient (numbers above)

reg [15:0] quotient;

reg [31:0] dividend\_copy, divider\_copy, diff;

wire [15:0] **remainder** = dividend\_copy[15:0];

reg [4:0] bit;

wire **ready** = !bit;

initial bit = 0;

always @( posedge clk )

if( ready && start ) begin

bit = 16;

quotient = 0;

dividend\_copy = {16'd0,dividend};

divider\_copy = {1'b0,divider,15'd0};

end else begin

diff = dividend\_copy - divider\_copy;

quotient = quotient << 1;

if( !diff[31] ) begin

dividend\_copy = diff;

quotient[0] = 1'd1;

end

divider\_copy = divider\_copy >> 1;

bit = bit - 1;

end

endmodule

// :**Example**:

//

// Streamlined divider.

// Based on :PH: Figure 4.41

//

// Uses less register storage than simple divider.

module **streamlined\_divider**(quotient,remainder,ready,dividend,divider,start,clk);

input [15:0] dividend,divider;

input start, clk;

output quotient,remainder;

output ready;

reg [31:0] qr;

reg [16:0] diff;

//

// 0000 1011

// """"""""|

// 1011 | 0001 0110 <- qr reg

// -0011 | -0011 <- divider (never changes)

// """"""""|

// 1011 | 0010 110o <- qr reg

// -0011 | -0011

// """"""""|

// 1011 | 0101 10oo <- qr reg

// -0011 | -0011

// """"""""| 0010 1000 <- qr reg before shift

// 0101 | 0101 0ooi <- after shift

// -0011 | -0011

// """"""""| 0010 ooii

// 10 |

//

// Quotient, 3 (0011); remainder 2 (10).

wire [15:0] **remainder** = qr[31:16];

wire [15:0] **quotient** = qr[15:0];

reg [4:0] bit;

wire **ready** = !bit;

initial bit = 0;

always @( posedge clk )

if( ready && start ) begin

bit = 16;

qr = {16'd0,dividend};

end else begin

diff = qr[31:15] - {1'b0,divider};

if( diff[16] )

qr = {qr[30:0],1'd0};

else

qr = {diff[15:0],qr[14:0],1'd1};

bit = bit - 1;

end

endmodule

////////////////////////////////////////////////////////////////////////////////

/// **"Real" Arithmetic Circuits**

// :HP: Appendix A

/// **Adders**

//

// Multiple-level CLA covered in class reasonably close to adders used

// in processors.

//

// Other fast adder techniques also used. (See :PH: problems.)

/// **Multipliers**

//

// The multiplication hardware presented above is much slower than

// the hardware used in real processors.

//

// "Real" n-bit Multiplier Features

//

// Multiplication done in one or two cycles (assume one cycle).

// Uses higher-radix (say 4) Booth recoding or something similar.

// Enough adders are provided so that product computed in one cycle.

// These adders can add n numbers quickly without the cost of n CLA's.

// Cost may be something like 3/2 n ripple adders plus 1 CLA.

// Delay through each adder is only a handful of gates.

// Adders may be connected as a tree, so that number of adders

// from input to output is proportional to log n (rather than n).

/// **Dividers**

//

// The division hardware presented above is slower than hardware used

// in real processors.

//

// Division trickier than multiplication because result of step i

// needed for i+1. This precludes the tree structures used in

// fast multipliers.

//

// FP Dividers in general-purpose processors typically take 10-20 cycles.

//

// Several division techniques used. Examples:

// SRT: Sweeney, Robertson, Tocher

// Newton Iteration

// Goldschmidt's Algorithm

//