CS220 Lab#4 Adder/Subtractor, Divider, Multiplier

Mainak Chaudhuri
Indian Institute of Technology Kanpur

Sketch

- Assignment#1: 32-bit Adder/Subtractor
  - New learning: combinational logic to do addition/subtraction of two 2's complement numbers
  - Carries two marks
- Assignment#2: 32-bit divider and multiplier
  - New learning: how to implement iterative algorithms in hardware
  - Part-A (32-bit divider): carries three marks
  - Part-B (32-bit multiplier): carries three marks

Assignment#1

- Design a 32-bit adder/subtractor
  - Refer to the lecture slides on number representation for the design
  - Takes three inputs: 32-bit *a*, 32-bit *b*, and one-bit *sub*
    - Both *a* and *b* are in 2's complement representation
    - Computes $a + b$ if *sub* is zero; otherwise computes $a - b$
  - Outputs a 32-bit result
    - The result should be interpreted as a 2's complement number
    - Recall that carry output should be ignored

Assignment#1

- Design a 32-bit adder/subtractor
  - Do behavioral simulation to test functional correctness
  - Create AXI4 IP, synthesize, implement, generate bit stream, export hardware as done in previous assignments
  - Show that your design works by writing a C program in Vitis IDE
    - Declare your variables in the C program as *int* so that you can specify negative values as well
    - Negative values are automatically represented in 2's complement by the PS; so no additional conversion is needed before writing the values of *a* and *b* to *slv_regs* or before printing the result (make sure to use %d as the format specifier in *xil_printf*)

Assignment#2A

- Design a 32-bit divider
  - We will assume that both divisor and dividend are unsigned 32-bit integers
  - The division algorithm that you will implement iteratively subtracts the divisor from the dividend until the result becomes negative
  - One simple way of implementing iterative algorithms is to have a clock input and do one iteration in each clock cycle
    - In this case, one iteration is one 32-bit subtraction
    - Subtraction result of the current clock cycle becomes the new dividend for the next clock cycle
    - Divisor for all clock cycles is the same

Assignment#2A

- Design a 32-bit divider
  - Make use of a 32-bit adder/subtractor to do subtraction
  - To easily detect the stopping criterion, modify the adder/subtractor module as follows
    - Output the sign bit of the result separately
    - If the result is negative, output *a* as the result as opposed to a – b (this avoids doing an extra addition at the end to calculate the remainder)
  - Quotient is one less than the number of clock cycles needed to get to the stopping criterion
    - Need to have a 32-bit counter for this purpose
    - The counter must be a register because it needs to remember the count so far

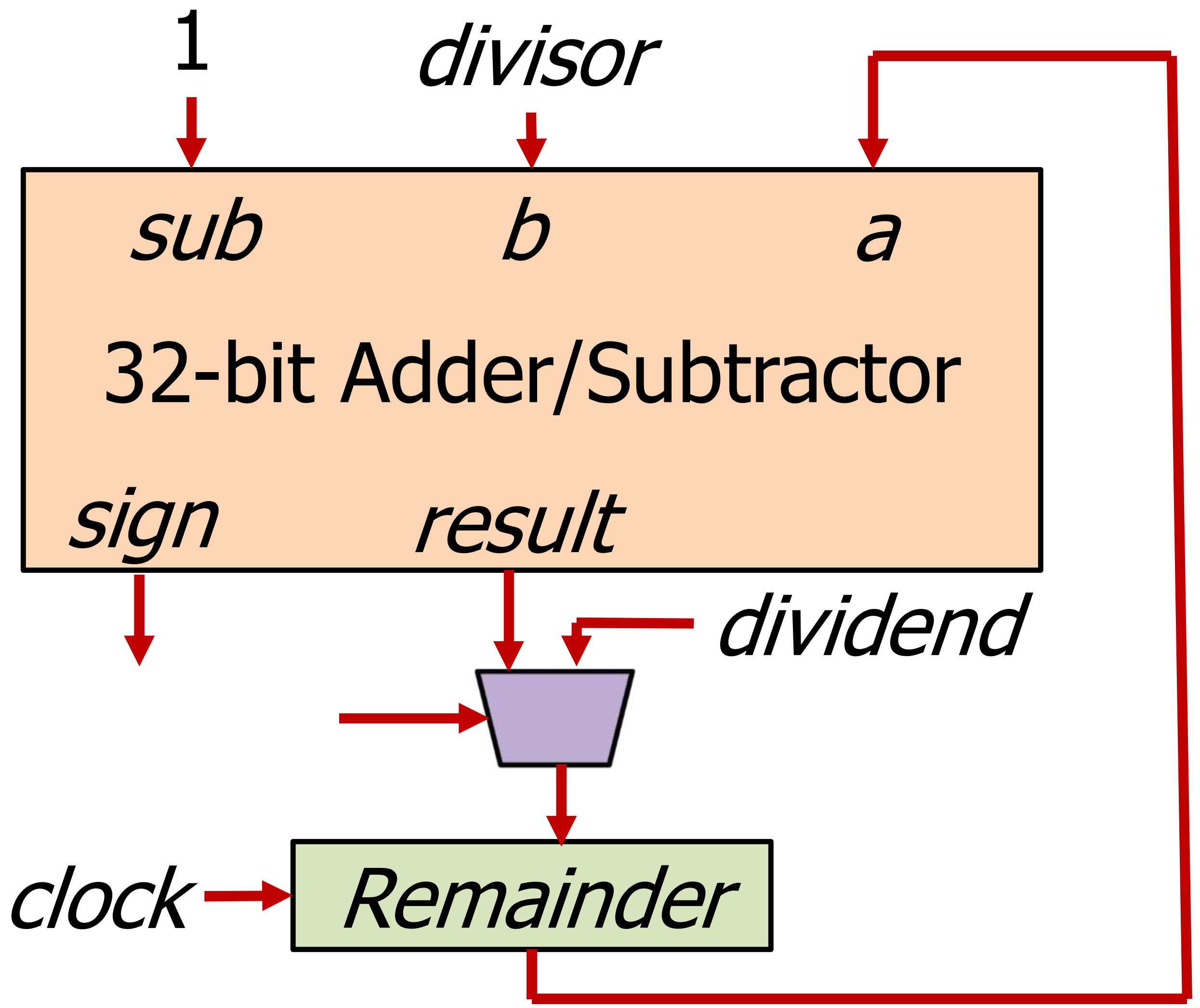
Assignment#2A

- Design a 32-bit divider
  - The top-level 32-bit divider module takes four inputs
    - 32-bit dividend, 32-bit divisor, one-bit reset, one-bit clock
    - The reset input should be used to reset the state of the divider before a division operation begins
    - A division operation begins only when reset is 0
  - The top-level 32-bit divider module has three outputs
    - 32-bit quotient, 32-bit remainder, one-bit done
    - The done output becomes 1 when a division operation ends; this will be used by the environment to find out when it is safe to read the quotient and remainder

Assignment#2A

- Design a 32-bit divider
  - The basic schematic is shown below

Remainder is a register



Assignment#2A

- Design a 32-bit divider
  - When the divisor is 0, you can output remainder equal to 0 and quotient equal to 2^{32}-1 even before doing any subtraction
  - When divisor is 1, you can output remainder equal to 0 and quotient equal to dividend even before doing any subtraction
- Do behavioral simulation to check functional correctness

Assignment#2A

- Create AXI4 IP, synthesize, implement, generate bit stream, export hardware
  - In the AXI module, pass *S_AXI_ACLK* as the input clock to the instantiation of your divider module
- Show that the divider works by writing a C program using Vitis IDE
  - The variables in your C program should be declared *unsigned*
  - Among the test cases, try the one that is expected to take the longest i.e., dividend = $2^{32} - 1$ and divisor = 2

Assignment#2A

- The structure of the C code that you will add

```
unsigned dividend, divisor, reset=1, quotient,
remainder, done=0;
Xil_Out32(base_addr+2, reset); // Reset divider
dividend = (1ULL << 32) – 1;
divisor = 2;
reset = 0;
Xil_Out32(base_addr, dividend);
Xil_Out32(base_addr+1, divisor);
Xil_Out32(base_addr+2, reset);
while (!done) done = Xil_In32(base_addr+5);
quotient = Xil_In32(base_addr+3);
remainder = Xil_In32(base_addr+4);
xil_printf(“%u/%u = %u, %u mod %u = %u\n”,
dividend, divisor, quotient, dividend, divisor, remainder);
```

Assignment#2B

- Design a 32-bit multiplier
  - The algorithm is to iteratively add the multiplicand
  - Do one addition in each clock cycle
  - Number of clock cycles needed is one less than the magnitude of the multiplier
    - If multiplier is -1, 0, or 1, no addition is needed
  - Since the result can be 64 bits, we will make use of the 64-bit adder developed in the previous lab
  - Your implementation should handle negative multiplicand and multiplier represented in 2’s complement
    - The 64-bit adder will correctly add two 2’s complement numbers without any change

Assignment#2B

- Design a 32-bit multiplier
  - The easiest way to handle negative multiplicand or negative multiplier is to
    - leave the multiplicand as it is
    - compute the magnitude of the multiplier
      - XOR the most significant bit with all bits and add the most significant bit: |x| = (x^{32{x[31]}})+{31’b0,x[31]}
    - use the magnitude of the multiplier to count the number of cycles needed to complete the multiplication
    - negate the final result if the multiplier is negative
      - If the final 64-bit result is y and the multiplier is x, then the output result should be (y^{64{x[31]}})+{63’b0,x[31]}

Assignment#2B

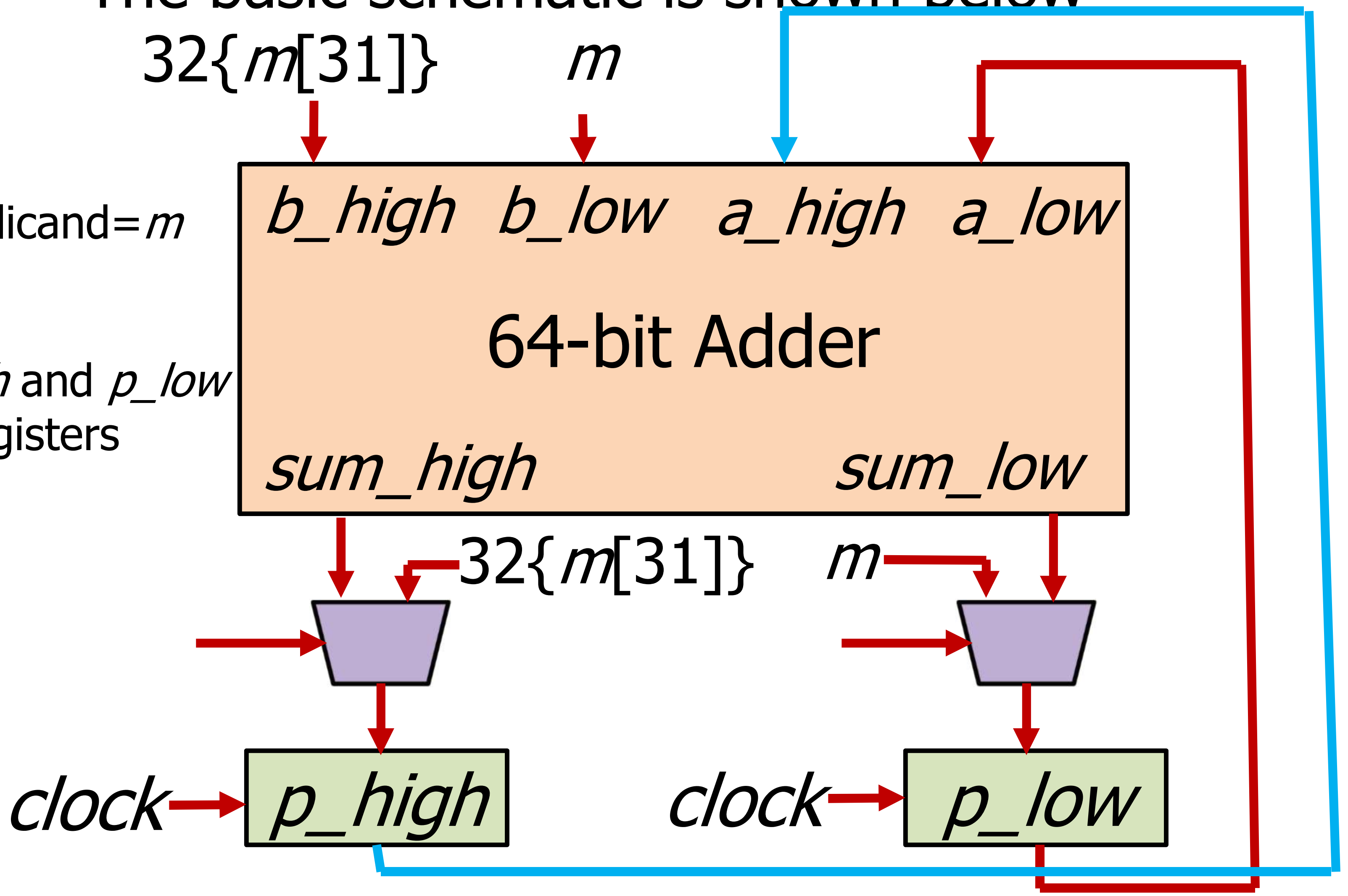
- Design a 32-bit multiplier
  - The top-level multiplier module takes four inputs
    - 32-bit multiplicand, 32-bit multiplier, one-bit reset, one-bit clock
    - The multiplicand and multiplier are expected to be in 2’s complement representation
  - The top-level multiplier module produces three outputs
    - 32-bit product_high, 32-bit product_low, one-bit done
      - product_high is the most significant 32 bits of the result
      - product_low is the least significant 32 bits of the result
    - {product_high,product_low} is interpreted as a 64-bit 2’s complement number

Assignment#2B

- Design a 32-bit multiplier
  - The basic schematic is shown below

multiplicand=*m*

p_high and *p_low* are registers



Assignment#2B

- Do behavioral simulation to check functional correctness
- Create AXI4 IP, synthesize, implement, generate bit stream, export hardware
  - In the AXI module, pass *S_AXI_ACLK* as the input clock to the instantiation of your multiplier module

Assignment#2B

- Show that the multiplier works by writing a C program using Vitis IDE
  - The variables in your C program should be declared *int* so that you can send negative as well as non-negative inputs (use %d to print)
  - Try out multiplicand and multiplier of both signs
  - Try largest positive multiplicand and multiplier
    - Multiplicand=multiplier=$2^{31} – 1$
  - Try the largest magnitude negative multiplicand and multiplier
    - Multiplicand=multiplier=-2^{31}
  - Combination of these and other cases