CS47 - Lecture 19

Kaushik Patra
(kaushik.patra@sjsu.edu)

1

- Multiplication

Reference Books:

1) *Chapter 3 of 'Computer Organization & Design by Hennessy, Patterson*

Multiplication Operation ...

2

Paper-Pencil Binary Multiplication

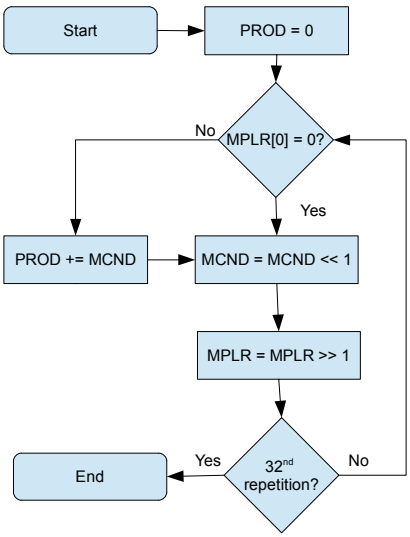
```
      1000   <- Multiplicand
    X 1001   <- Multiplier
-------------
      1000
+    00000
+   000000
+ 1000000
-------------
  1001000   <- Product
```

- First operand is the multiplicand and the second operand is multiplier.
- n-bit x m-bit multiplication will produce (n+m)-bit result.
- At each step of multiplication the multiplicand needs to be placed at the right place if multiplier bit is 1, all zeros otherwise.
- Final product is sum of all the steps.

3

- Basic rule for binary multiplication is A * 1 = A and A * 0 = 0.
- Partial products are computed at each steps. Partial products are shifted by number of steps to get final partial product. For example if X is the partial product at step n then final partial product will be (X << (n-1)).
- At the end of the procedure of partial product computation all the partial products are added to get the final product.

Binary Multiplication Algorithm



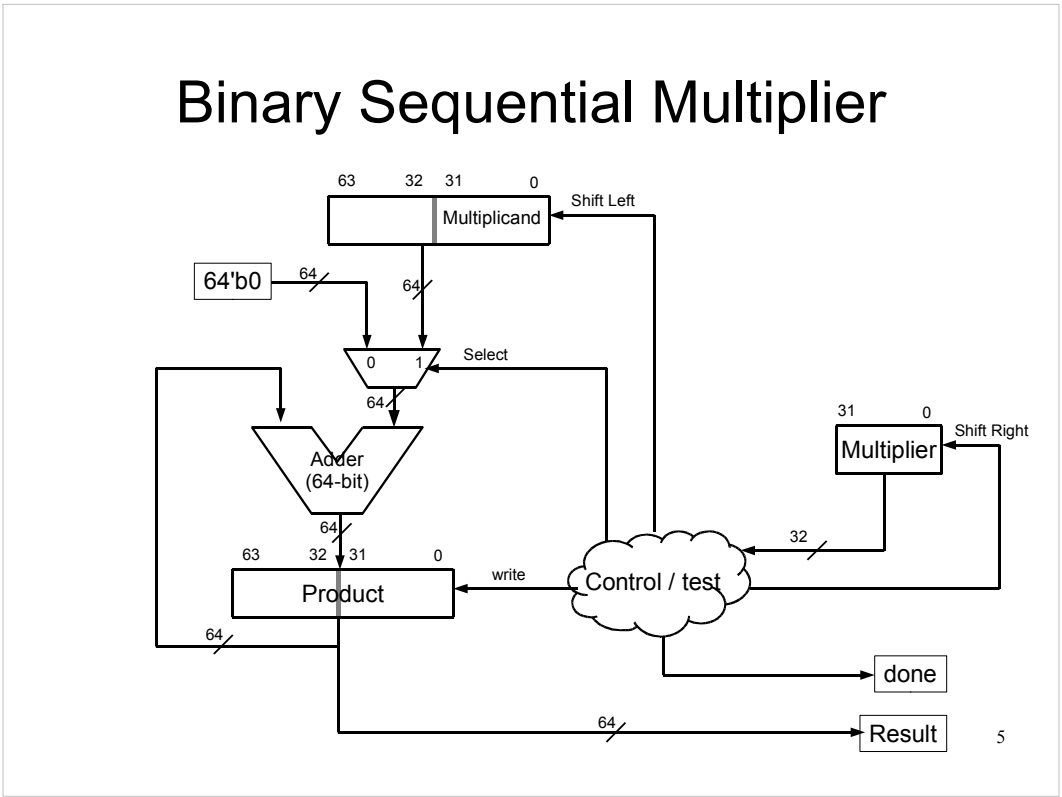
MCND – Multiplicand

MPLR – Multiplier

PROD - Product

4

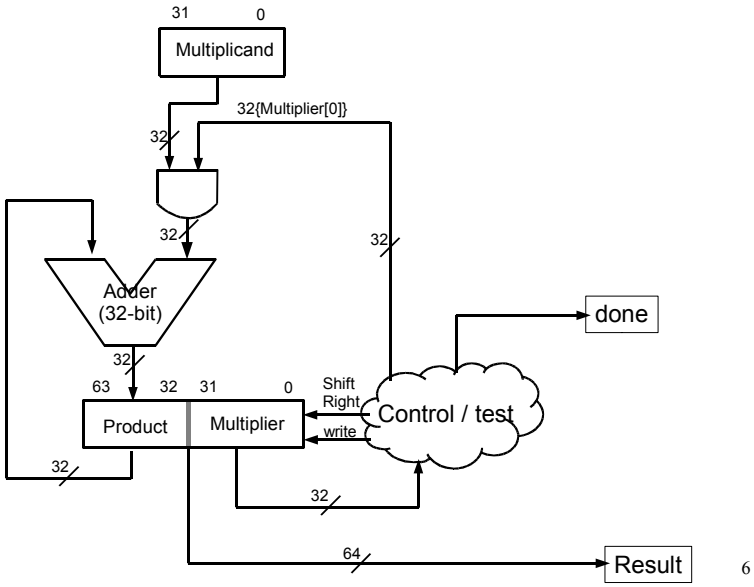
- First the product result PROD is set to 0.
- For a 32-bit multiplier, for 31 steps
  - LSB of multiplier MPLR is tested and if it is 1 then PROD is added with the multiplicand MCND.
  - The very next step is to shift MCND to left by 1 bit and MPLR is shifted right by 1 bit.
- We could make the operation faster by testing is the MPLR is all 0 and stop the operation. However, this will result is variable completion time for multiplication operation and need to have an extra 'DONE' signal for controller to know that multiplication is done.



- The lower half of the 64-bit multiplicand register is loaded with the 32-bit multiplicand. Multiplier is loaded into a 32-bit register. A 64-bit 2x1 mux is used to select between all 0 or the multiplicand register value depending on the LSB value of the multiplier register. This mux output is fed into a 64-bit adder. The output of the adder is stored in a 64-bit output product register which is initialized to 0 at start of the operation. The register value is also fed as the other input of the adder.

- At each clock cycle, the controller issues the following control signal in sequence.
  - mux selection signal based on the LSB value of the multiplier.
  - Then it issues write signal for the product register so that it can store partial product and sum till the step.
  - Then it issues left shift signal to multiplicand register and right shift signal to the multiplier register.
  - It repeat the steps from mux selection signal for 32 times.
  - Once done, it sends out optional DONE signal and the product register value as result.

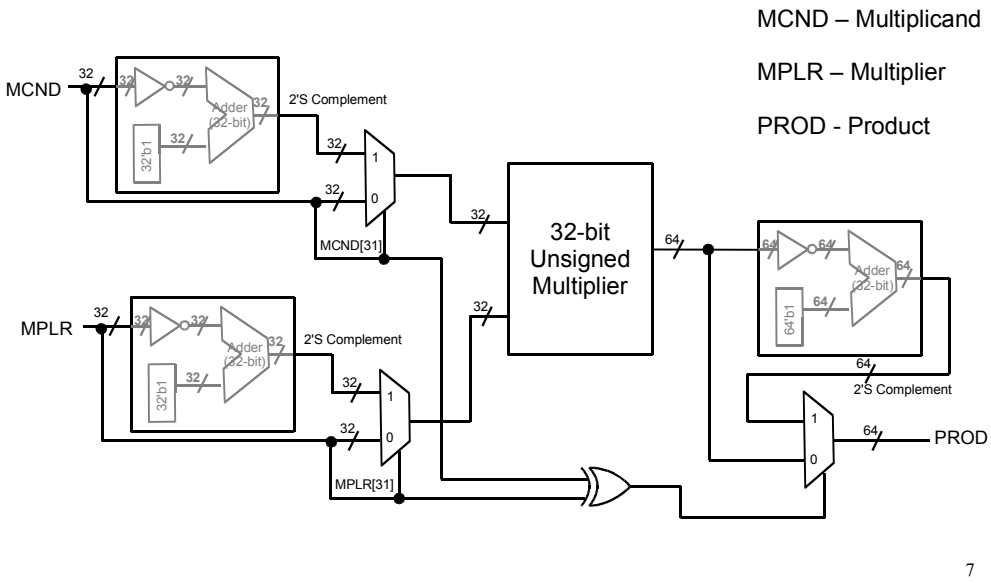
Simplified Sequential Multiplier



- The previous circuit can be simplified to a great extent as following.
  - Use 32-bit static register (no shift capability) for multiplicand.
  - Use 32-bit adder. It loads the upper half of the 64-bit product register.
  - Use 64-bit product register, but use its lower half to load the multiplier and upper half is reset to 0. At each step multiplier needs 1 fewer bits and product will grow by 1 bit. Hence they both can be accommodate within a single 64-bit register by space sharing. This strategy works because shifting the multiplicand by 1 bit to left is equivalent to shifting the product
  - Substitute MUX with AND. One of the input for the AND is the multiplicand and the other input is the 32-bit replication of multiplier LSB.

- At each clock cycle control signal does the following for 32 times.
  - Issue write signal to load the upper half of the product register with the adder output.
  - Issue right shift signal to shift both upper half of the product register (the adder result) and lower half of the product register (the multiplier).
  - Once done, all 64-bit contains the product result and the optional DONE signal is issued.

Signed Multiplication Circuit



- The multiplication considered so far is unsigned multiplication. To implement signed multiplication we need to do the following.
  - Determine if the operand is -ve.
  - Any -ve operand will be translated to +ve value, but the sign will be remembered.
  - Perform multiplication as unsigned operation.
  - If sign of multiplicand and multiplier are same then take the product as it is, convert to corresponding -ve number otherwise.
  - Need look out for overflow condition.

- If we assume that numbers are represented in 2's complement format, we can convert the numbers from -ve to +ve and vice-verse using 2's complement converter. The converter implements (A'+1) functionality which is the formula to convert +ve number in 2's complement format to its -ve value and vice-versa.

- Depending on the sign bit (MSB) of the input numbers, its original value or complemented value (incase it is -ve) is fed into the unsigned multiplier.

- If the sign bit (MSB) of two number agrees (detected by XOR operation between two sign bit of the two number) then we select the output of the unsigned multiplier as output. The complemented form is selected otherwise.

Multiplication in MIPS Assembly ...

8

Utility Procedures

- Create a procedure 'twos_complement'
  - Arguments
    - $a0 : Number of which 2's complement to be computed
  - Return
    - $v0 : Two's complement of $a0
  - Use 'add_logical' and 'not' to compute '~$a0 + 1'
  - Approx 20 line of code including frame store/restore

Utility Procedures

- Create a procedure 'twos_complement_if_neg'
  - Arguments
    - $a0 : Number of which 2's complement to be computed
  - Return
    - $v0 : Two's complement of $a0 of $a0 is negative
  - Test $a0 value less than 0 and use 'twos_complement' if needed
  - Approx 20 line of code including frame store/restore

Utility Procedures

- Create a procedure 'twos_complement_64bit'
  - Arguments
    - $a0 : Lo of the number
    - $a1 : Hi of the number
  - Return
    - $v0 : Lo part of 2's complemented 64 bit
    - $v1 : Hi part of 2's complemented 64 bit
  - Upgrade add_logical to return final carryout in $v1
  - Steps (approx 30 line of code including frame handling)
    - Invert both $a0, $a0.
    - Use add_logical to add 1 to $a0
    - Use add_logical to add carry from previous step to $a1

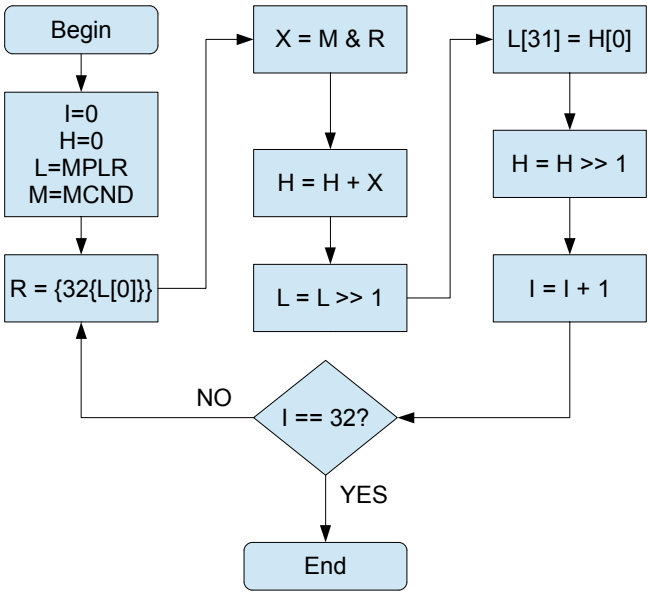
Utility Procedures

- Create a procedure 'bit_replicator' two replicate given bit value to 32 times.
  - Arguments
    - $a0 : 0x0 or 0x1 (the bit value to be replicated)
  - Return
    - $v0 : 0x00000000 if $a0 = 0x0
    - 0XFFFFFFFF if $a0 = 0x1
  - Approx 20 line of code

Unsigned Multiplication

- Create a procedure 'mul_unsigned'
  - Arguments
    - $a0 : Multiplicand
    - $a1 : multiplier
  - Return
    - $v0 : Lo part of result
    - $v1 : Hi part of result
  - Approx 45 line of code

Unsigned Multiplication



14

- Use replication procedure to replicate single bit value to 32 bits to determine R. Use mask to determine H[0] and then use 'insert_to_nth_nit' macro to insert this bit into L[31].

Signed Multiplication

- Create a procedure 'mul_signed'
  - Arguments
    - $a0 : Multiplicand
    - $a1 : multiplier
  - Return
    - $v0 : Lo part of result
    - $v1 : Hi part of result
  - Approx 50 line of code

Signed Multiplication

- Steps
  - N1 = $a0, N2 = $a1
  - Make N1 two's complement if negative
  - Make N2 two's complement if negative
  - Call unsigned multiplication using N1, N2. Say the result is Rhi, Rlo
  - Determine sign S of result
    - Extract $a0[31] and $a1[31] bits and xor between them. The xor result is S.
  - If S is 1, use the 'twos_complement_64bit' to determine two complement form of 64 bit number in Rhi, Rlo.

CS47 - Lecture 19

Kaushik Patra
(kaushik.patra@sjsu.edu)

17

- Multiplication

Reference Books:

1) *Chapter 3 of 'Computer Organization & Design by Hennessy, Patterson*