Arithmetic Algorithms and ALU Design

Mainak Chaudhuri
Indian Institute of Technology Kanpur

Sketch

- Abstract model of computer
- ALU architecture
- Arithmetic algorithms
  - Overflow detection in addition and subtraction
  - Integer multiplication
  - Integer division
  - Floating-point addition
  - Floating-point multiplication

Abstract model of computer

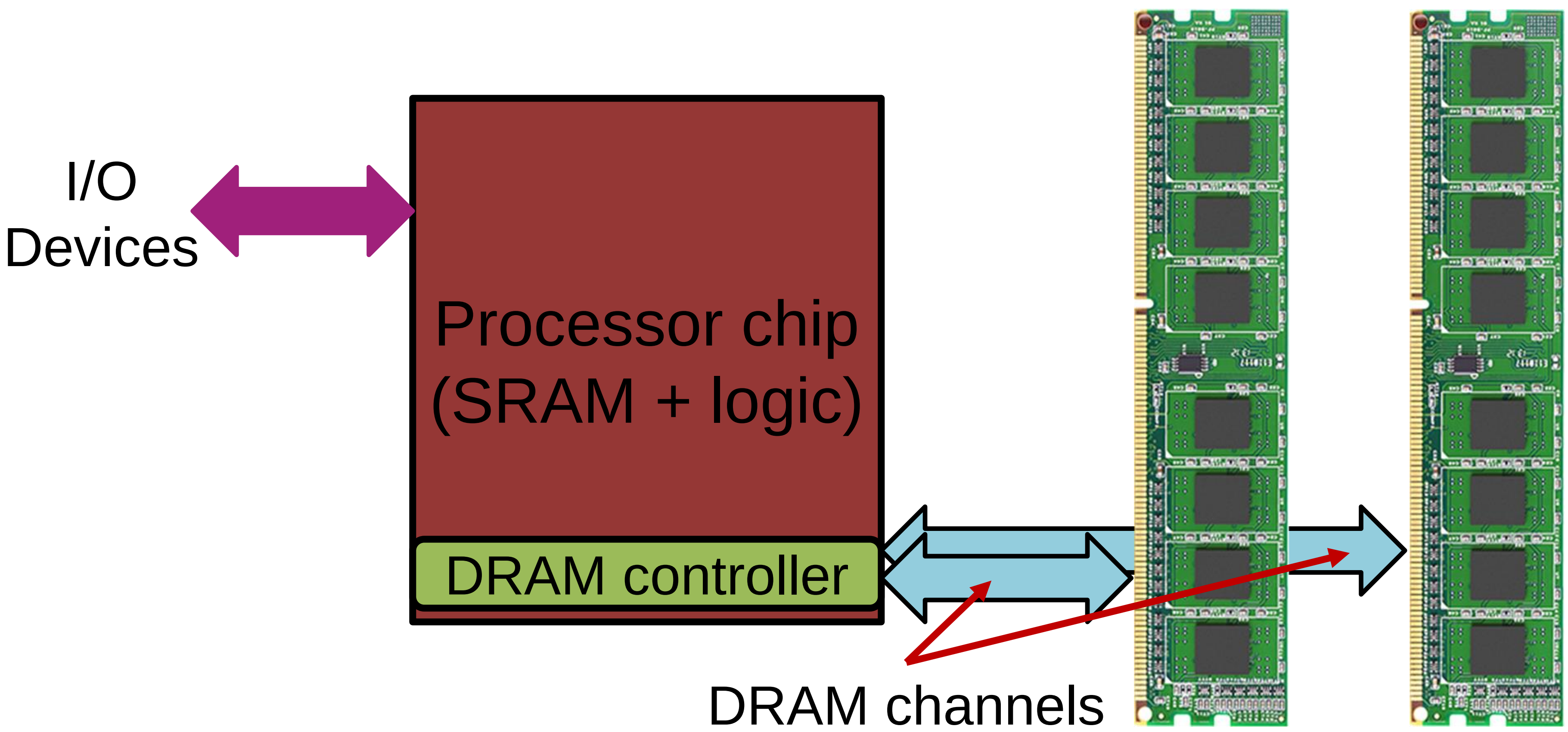
- Computer has an ISA
- The implementation of the ISA is an abstract five-state synchronous FSM
  - Each state change happens on posedge clock
  - State 0: fetch the instruction pointed to by program counter from memory; update program counter to point to the next instruction
  - State 1: decode the instruction to extract various fields and read source register operands
  - State 2: execute the instruction in ALU; compute address of load/store instructions; update program counter if control transfer instruction
  - State 3: access memory if load/store instruction
  - State 4: write result to destination register if the instruction produces a result

Abstract model of computer

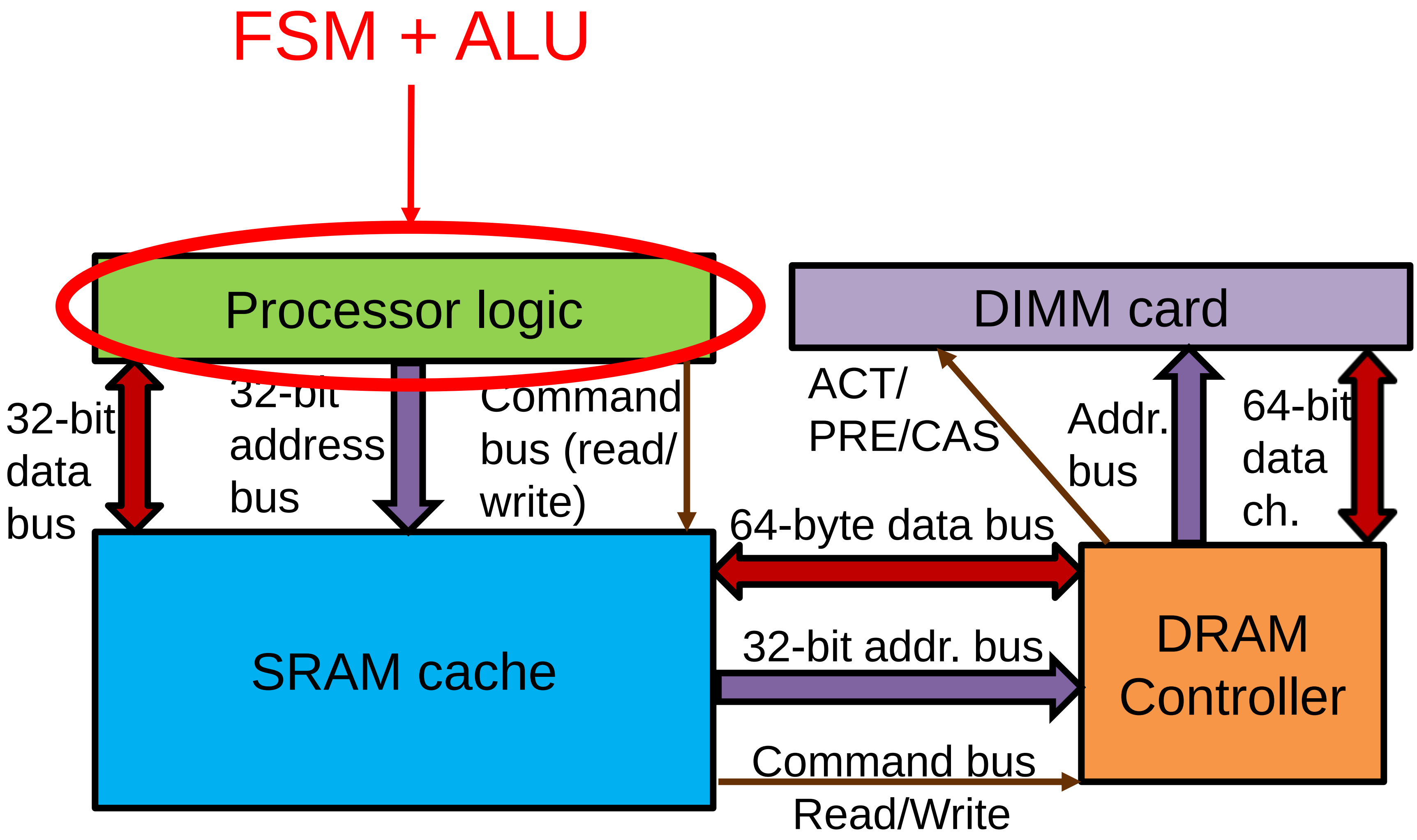
- Fetching an instruction requires accessing memory with the program counter as addr.
- Decoding an instruction for MIPS is simple due to small number of formats and fixed position of the register specifiers
- Reading operands from register file requires exercising the read ports
- Executing an instruction and computing address of a load/store instruction requires an arithmetic logic unit (ALU)
- Load/store instructions access memory with the computed address
- Writing result to register file requires exercising the write ports

Abstract model of computer

- The logic on processor chip implements the FSM and the ALU



Abstract model of computer



ALU architecture

- ALU is responsible for executing the core of the MIPS instructions
  - Everything except load/store instructions
- ALU takes two inputs for most instructions and produces a result that may or may not get written to a destination register
  - For example, control transfer instructions do not write to a general-purpose register, but writes to the program counter
- ALUs are of two kinds: integer and floating-point
  - The floating-point ALU is often referred to as the floating-point unit (FPU)

Integer ALU architecture

- Components of the integer ALU
  - Adder/subtractor
    - Executes add, addu, sub, subu, addi, addiu
    - Computes target of conditional branch instructions
    - Computes address of load/store instructions
  - Multiplier
    - Executes mult, multu
  - Divider
    - Executes div, divu
  - Logic gate array
    - Executes and, nor, or, xor, andi, ori, xori, lui
  - Comparator and branch subunit
    - Executes slt, sltu, slti, sltiu
    - Executes the comparison part of beq, bne, bgez, bgtz, bltz, blez

Integer ALU architecture

- Components of the integer ALU
  - Shifter
    - Executes sll, sllv, sra, srav, srl, srlv
    - Naïve implementation would shift by one bit position in every clock cycle
      - Inefficient for arbitrary shift amounts
    - Barrel shifters can shift an operand by arbitrary amount in time no more than an addition takes
  - Multiplexer (part of branch subunit)
    - Selects between branch target and PC+4 depending on the output of the comparison part of cond. branch
  - Data movement unit (part of multiplier/divider)
    - Executes mflo, mfhi, mtlo, mthi
  - Some designs do not consider the shifter, multiplier, and divider as part of the ALU

Overflow detection in MIPS

- MIPS ISA offers two types of arithmetic instructions for addition and subtraction
  - One type detects overflows (add, sub, addi) and the other doesn't (addu, subu, addiu)
  - Whether overflow should be detected in arithmetic operations is a part of the HLL specification
    - MIPS ISA can support both types of HLLs
  - If add/sub overflow is supposed to be detected, MIPS raises an exception on arithmetic overflow
    - Saves PC in EPC
    - Saves all general-purpose and floating-point registers in memory
    - Jumps to a fixed handler which examines cause and status registers to invoke the appropriate exception handler

Overflow detection in MIPS

- Arithmetic overflow exception
  - On completion of the handler
    - Restore all general-purpose and floating-point registers from memory
    - Compute EPC+4 and move it to \$k0 or \$k1 (same as \$26 and \$27)
      - These two registers are reserved for use of operating system (e.g., interrupt, exception, system call handlers)
      - These two registers are not preserved across handlers
    - Execute jr \$k0 or jr \$k1 depending on where EPC+4 is
- Overflow detection is also possible in software
  - Not very relevant for MIPS addition/subtraction
  - Important for processors that cannot detect overflow in hardware

Overflow detection in software

- Software cannot access the carry bits into and out of the most significant bit position
  - So, cannot compare these to detect overflow
- Observation
  - In signed addition, an overflow is detected if and only if the operands have the same sign and the result has sign opposite of the operand
    - If the operands have opposite signs, the result is guaranteed to have smaller magnitude than the larger magnitude operand and hence, there cannot be an overflow
  - In signed subtraction, an overflow is detected if and only if the operands have opposite signs and the result has sign opposite of the first operand
    - If the operands have the same sign, the result is guaranteed to have smaller magnitude than the larger magnitude operand

Overflow detection in software

- Code for detecting overflow in signed addition ($t0 = $t1 + $t2)

```
addu $t0, $t1, $t2   # no overflow exception
xor  $t3, $t1, $t2
slt  $t3, $t3, $0    # $t3 is 1 iff signs of $t1, $t2 differ
bne  $t3, $0, no_overflow
xor  $t3, $t0, $t1
slt  $t3, $t3, $0    # $t3 is 1 iff signs of $t0, $t1 differ
bne  $t3, $0, overflow_handler

no_overflow: …
```

Overflow detection in software

- Unsigned addition and subtraction
  - Overflow is detected if and only if the magnitude of the result exceeds the maximum representable unsigned value
    - $2^{32} - 1$ for 32-bit processors
- Code for detecting overflow in unsigned addition ($t0 = $t1 + $t2)

```
addu $t0, $t1, $t2    # no overflow exception
nor   $t3, $t1, $0     # $t3 has 2^32 - $t1 – 1
sltu   $t3, $t3, $t2    # $t3 has 1 iff $t1 + $t2 > 2^32 – 1
bne  $t3, $0, overflow_handler
```

  - Note that the comparison must be unsigned (sltu) so that 2^{32} - $t1 – 1 and $t2 are treated as unsigned operands

Multiplication algorithm

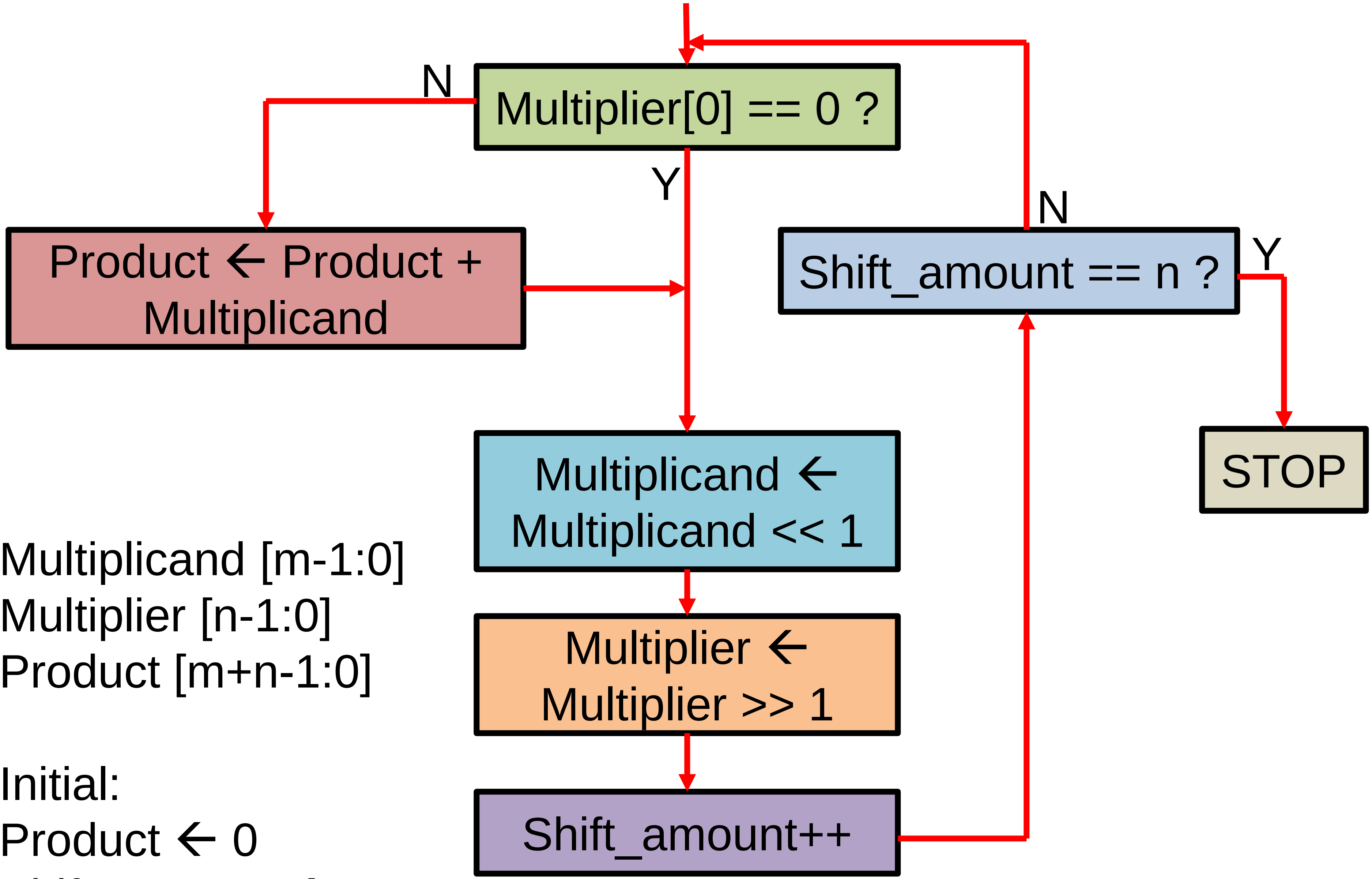
- One operand is called the multiplicand and the other is called the multiplier
  - m-bit multiplicand and n-bit multiplier lead to (m+n)-bit product
- Paper-pencil algorithm (assume unsigned operands)

```
      1001   multiplicand
    x 1010   multiplier
---------------------------------------
      0000
     1001    (why shift left?)
    0000
   1001
---------------------------------------
   1011010
```

Multiplication algorithm

- Paper-pencil algorithm (unsigned operands)
  - Multiplicand = $(a_{m-1}a_{m-2}\ldots a_0)$ = A
  - Multiplier = $(b_{n-1}b_{n-2}\ldots b_0) = \sum_{i=0}^{n-1} b_i 2^i$
  - Product = $A.b_0 + A.b_1.2 + A.b_2.2^2 + \ldots$
  - The left shift operations are needed to take care of the multiplications by 2^i
  - The i^{th} partial product = (b_i ? A << i : 0)
  - Leads to a simple multiplication algorithm that in each step produces one partial product by inspecting a bit position of the multiplier and accumulates the partial product into a product register
  - We will refer to the i^{th} bit of multiplier as multiplier[i]

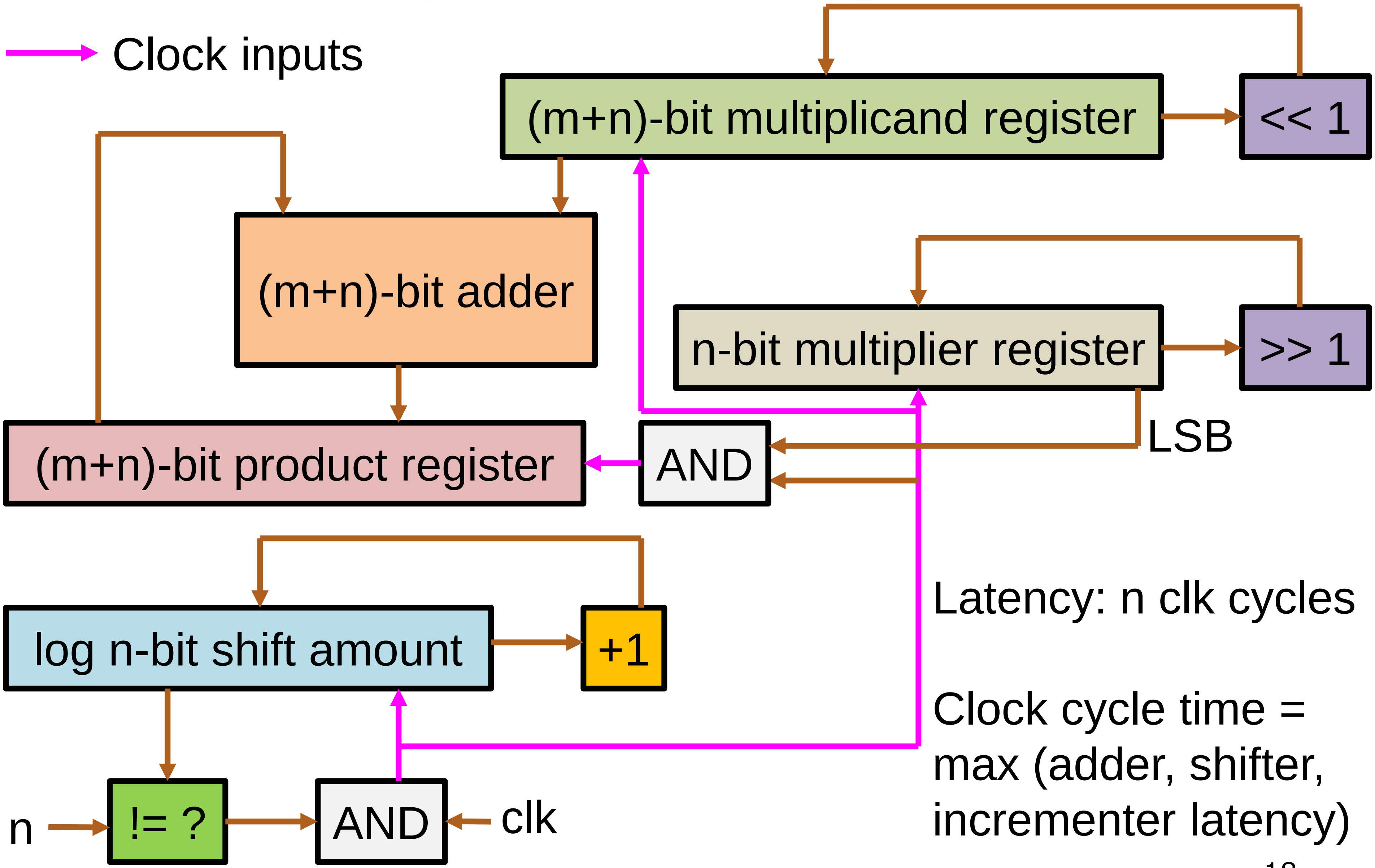
Multiplication algorithm



Multiplicand [m-1:0]
Multiplier [n-1:0]
Product [m+n-1:0]

Initial:
Product ← 0
Shift_amount ← 0

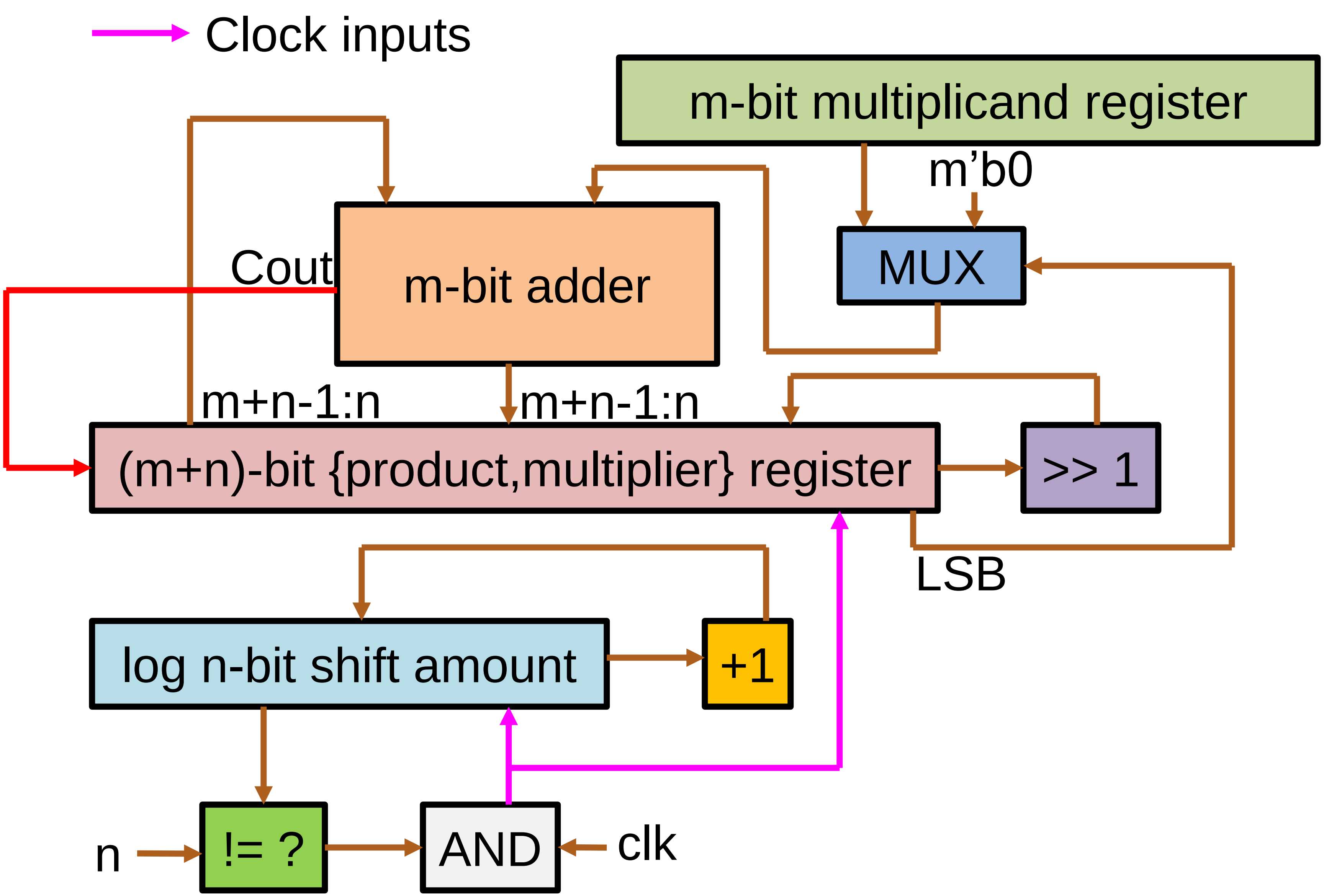
Multiplication algorithm



Latency: n clk cycles

Clock cycle time = max (adder, shifter, incrementer latency)

Multiplication algorithm (simpler)



Faster multiplication

- In the basic algorithm, n additions are done sequentially because we have used only one adder
- Consider the following grouping
  - Product = $(A.b_0+A.b_1.2) + (A.b_2.2^2+A.b_3.2^3) + (A.b_4.2^4+A.b_5.2^5) + \ldots$
  - The additions in the parentheses can be done in parallel after appropriately shifting the multiplicand (requires n/2 adders)
  - The results of these n/2 additions can be further paired and combined in n/4 adders
  - Overall, n – 1 additions would be arranged in a binary tree and the multiplication requires $\log_2 n$ cycles (can be done with n/2 adders)

Signed multiplication

- Negate the negative operands and do the multiplication
- If the signs of the operands are opposite, negate the multiplication result
- MIPS has both signed and unsigned multiplication
  - mult instruction treats operands as signed
  - multu treats operands as unsigned
- MIPS offers no hardware support for detecting overflow in multiplication
  - If needed, must be done in software

Booth's multiplication algorithm

- Due to Andrew Booth [1951]
- Handles both unsigned and signed operands
- Basic observation
  - Let the multiplicand be A and let the multiplier be $b_{n-1}b_{n-2}\ldots b_0$ = 000…011…100…0 where the block 1's ranges from b_p to b_q with $p < q$
  - Therefore, the multiplier is $2^{q+1} - 2^p$ and the product is (A << (q+1)) – (A << p)
  - Can be generalized by representing the multiplier as addition of disjoint blocks of 1's
- Append $b_{-1} = 0$ at the end of the multiplier
- Assume that both multiplicand and multiplier are in two's complement representation

Booth's multiplication algorithm

- Algorithm
  - Initialize product register to 0
  - Scan the multiplier from b_0 to b_{n-1}
  - When examining b_i, if $b_i == b_{i-1}$, keep scanning
  - When examining b_i, if b_i is 1 and b_{i-1} is 0, subtract (A << i) from product register
  - When examining b_i, if b_i is 0 and b_{i-1} is 1, add (A << i) to the product register
  - If i==n – 1, stop. Final product is in two's complement representation.

Booth’s multiplication algorithm

- Observations
  - In the worst case, needs n addition operations when the multiplier has an alternating bit pattern i.e., 0101…01
  - If b_{n-1} is 1 (negative multiplier), the last operation done on the product would be a subtraction
    - Will result in a positive product, if the multiplicand is also negative; otherwise the product is negative
- Booth’s algorithm can be implemented in two stages
  - First stage scans the multiplier and prepares the addition/subtraction operands
  - Second stage does the additions and subtractions in a binary tree of adders as in the fast multiplication algorithm

Division algorithm

- Suppose dividend (A) = ($a_{m-1}a_{m-2}\ldots a_0$), divisor (B) = ($b_{n-1}b_{n-2}\ldots b_0$)
- Problem statement: find Q and R such that A=BQ + R and R < B
  - We will assume unsigned A and B to start with
  - Clearly the maximum length of Q is m – n + 1 if m ≥ n; otherwise Q is 0 and R = A
  - Mathematically, we want

    $$\sum_{i=0}^{m-1} a_i 2^i = \sum_{i=0}^{n-1} b_i 2^i \sum_{i=0}^{m-n} q_i 2^i + \sum_{i=0}^{n-1} r_i 2^i$$

  - The usual division algorithm proceeds by computing q_i's starting from the most significant bit i.e., q_{m-n}

Division algorithm

- Steps in usual division algorithm
  - 1. Place A above B such that their most significant bits are aligned
    - This may require shifting B to the left by m – n positions (equivalent to multiplying B by 2^{m-n})
    - E.g., suppose A = 1001010, B = 1000 (m=7, n=4)

    A = 1001010

    B′ = 1000000 (shifted B)
  - 2a. If B′ is less than or equal to A, subtract B′ from A and assign Q = (Q << 1) | 1, A = the result of subtraction, B′ = B′ >> 1 (to compute next lower q_i; notice the similarity with polynomial division)
  - 2b. If B′ > A, shift B′ to the right by one bit position and assign Q = (Q << 1) [Q gets a 0 bit]
  - Repeat step 2a or 2b m – n + 1 times, each time producing one bit of Q; R is residual A at the end

Division algorithm

- Execution of the usual algorithm (iteration 1)

A = 1001010 Q = 0

B′ = 1000000

0001010 ← new A Q = 1

0100000 ← new B′

Division algorithm

- Execution of the usual algorithm (iteration 2)

A = 0001010 Q = 1

B′ = 0100000

No subtraction

0001010 ← A Q = 10

0010000 ← new B′

Division algorithm

- Execution of the usual algorithm (iteration 3)

A = 0001010 Q = 10

B′ = 0010000

No subtraction

0001010 ← A Q = 100

0001000 ← new B′

Division algorithm

- Execution of the usual algorithm (iteration 4)

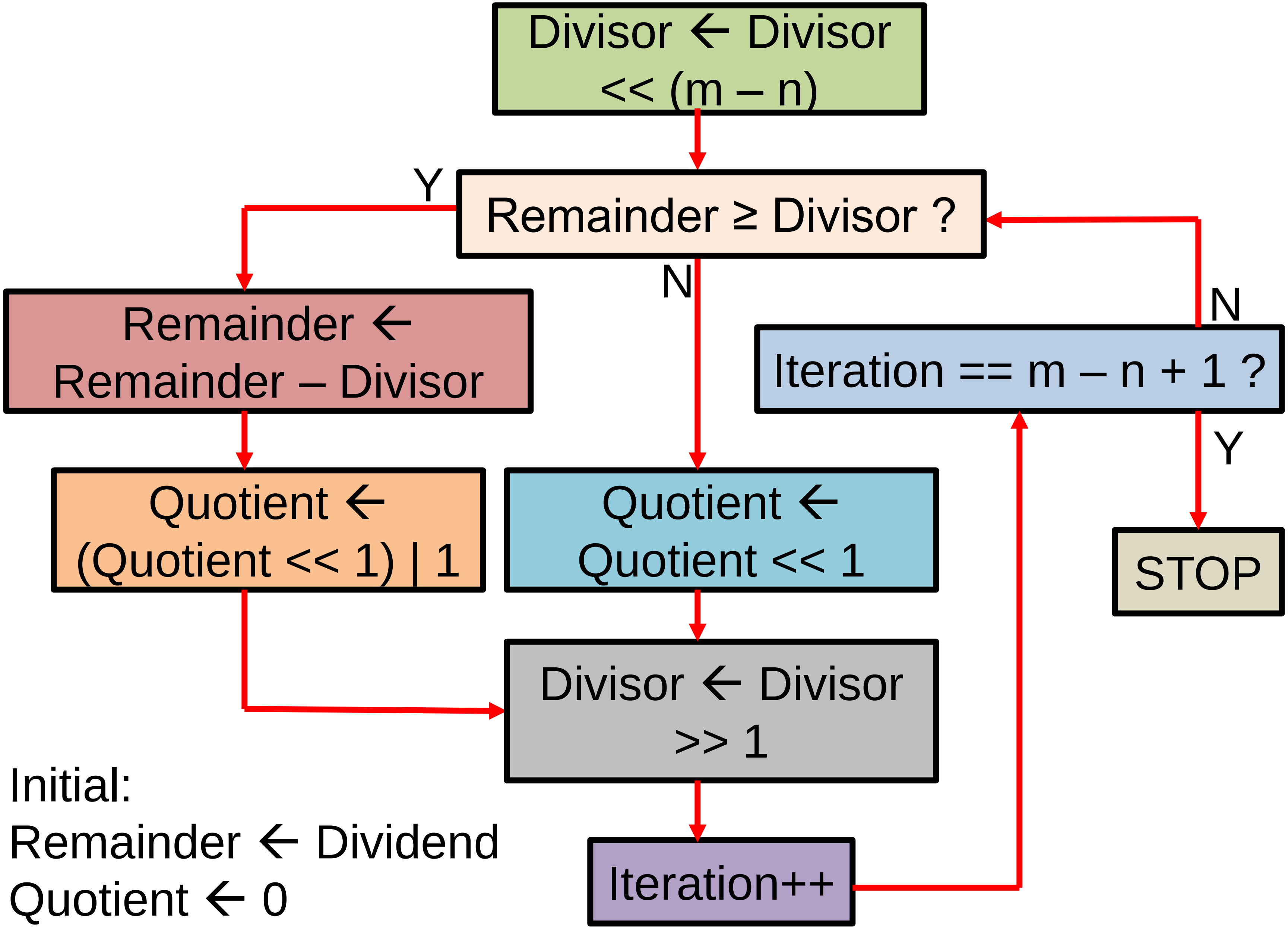
```
      A = 0001010                  Q = 100
      B′ = 0001000
   --------------------
           0000010  ← new A        Q = 1001
           0000100  ← new B′
```

  - Done executing four iterations ($m - n + 1$ iterations)
  - Quotient is 1001 and remainder is residual A i.e., 10
- Each iteration computes one coefficient q_i (either 0 or 1) of the polynomial $q_i 2^i$ by equating the leading powers of residual A and B

Division algorithm

- The first step of aligning the divisor and dividend is important for the main division algorithm to work correctly
  - In a processor, A and B will be in two 32-bit registers to begin with
  - The first step requires computing m – n and shifting B to the left by m – n bits
  - How to compute m and n quickly from A and B?
    - This is essentially a search problem that searches for the most significant non-zero bit in a given binary string

Division algorithm

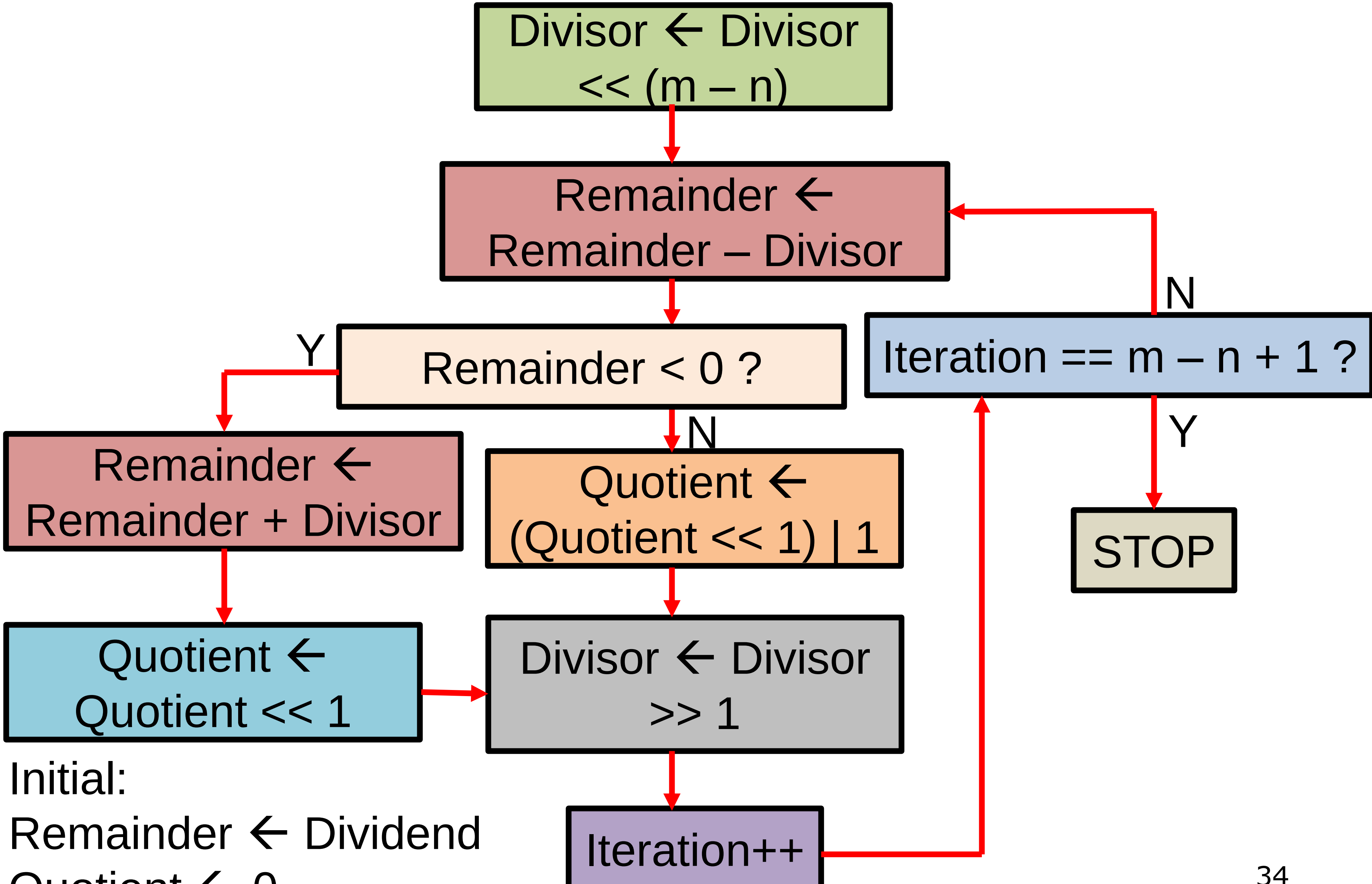


Initial:
Remainder ← Dividend
Quotient ← 0

Division algorithm

- Critical path of one iteration involves a comparison and a subtraction
  - Left shift and OR, right shift, increment can all be done in parallel with subtraction
- A variant of this algorithm always subtracts, then checks if the remainder is negative and if yes, restores the remainder
  - Known as restoring division algorithm
  - Checking if remainder is negative requires comparing only the most significant bit
  - Critical path now involves a subtraction, a negative check, and an addition

Restoring division algorithm

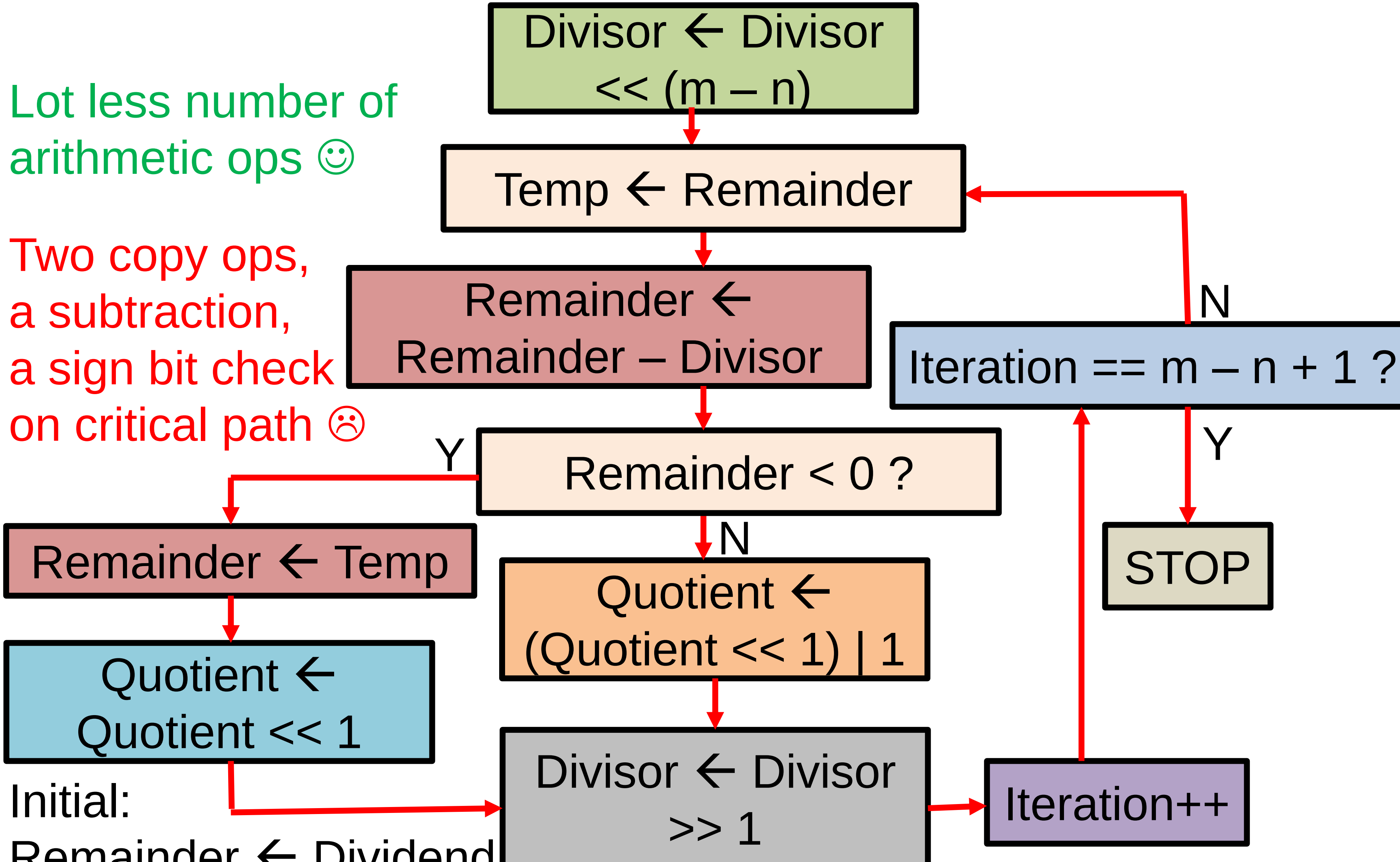


Initial:
Remainder ← Dividend
Quotient ← 0

Non-performing restoring division

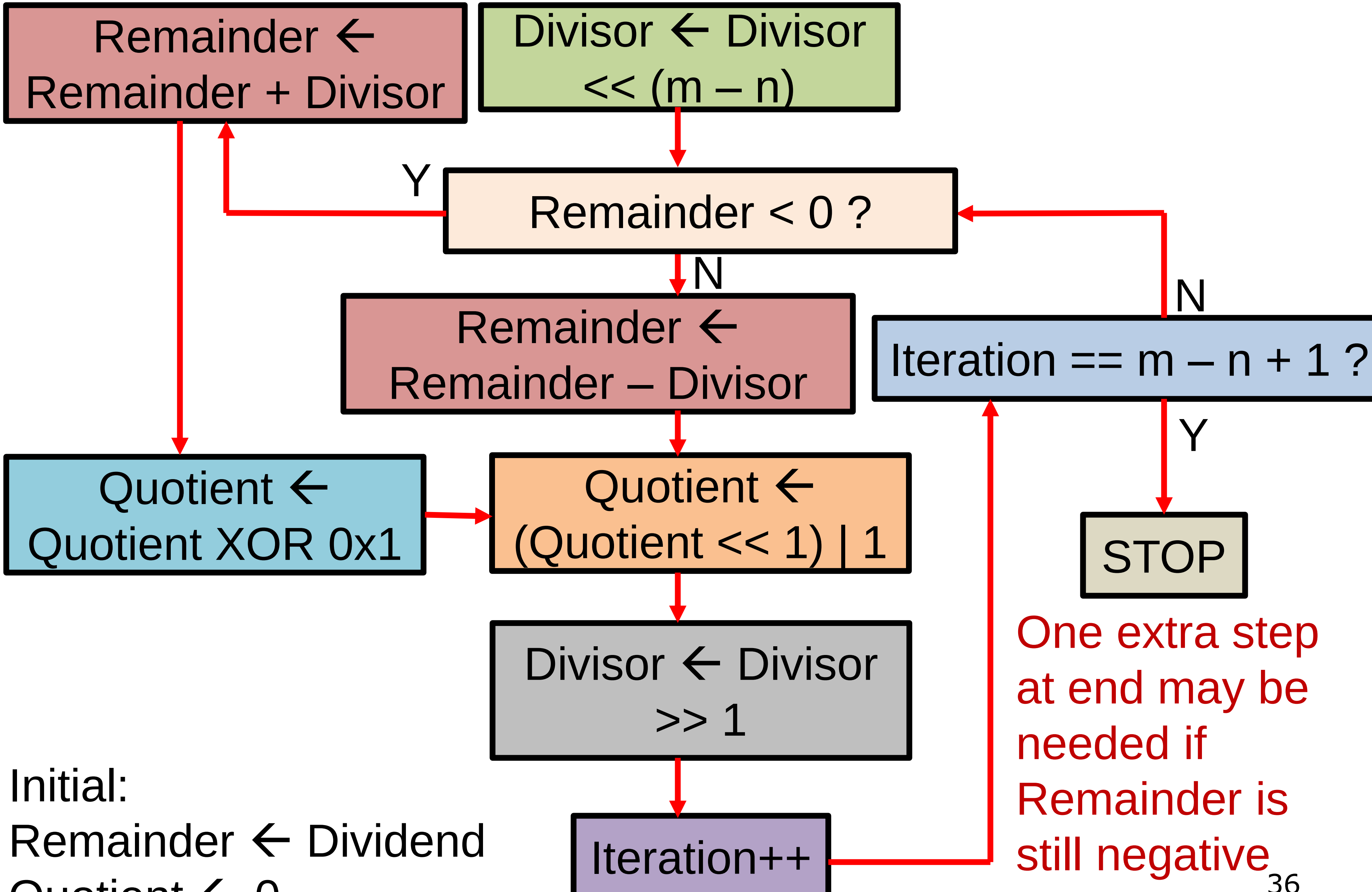
Lot less number of arithmetic ops ☺

Two copy ops,
a subtraction,
a sign bit check
on critical path ☹



Initial:
Remainder ← Dividend
Quotient ← 0

Non-restoring division algorithm



Initial:
Remainder ← Dividend
Quotient ← 0

One extra step at end may be needed if Remainder is still negative

Non-restoring division algorithm

- Why is this correct?
  - Suppose in iteration#k of restoring division, r – d turns out to be negative, so we restore r
  - In iteration#k+1 of restoring division, we compute r – d/2 (note that d is shifted right)
  - In non-restoring division's iteration#k, we keep r – d (which is negative) and in iteration#k+1, we do r – d + d/2 yielding r – d/2, which is same as the value of the remainder in restoring division after iteration#k+1
  - The quotient is adjusted in iteration#k+1 by first making the last shifted bit 0 and then shifting in a 1
- Critical path = sign bit check; (addition or subtraction || Quotient update || divisor shift || iteration increment) [";" → seq., "||" → parallel]

Non-restoring division algorithm

- Hardware implementation
  - Directly follows from the algorithm
  - It is possible to combine {remainder, quotient} in a single register of size m bits
    - At the beginning, the entire register is occupied by the m-bit dividend and quotient is zero
    - At the end, the least significant m – n bits along with the last quotient bit will be the final quotient and the remaining n bits will contain the remainder
    - Justifies why Lo contains quotient and Hi contains remainder in MIPS division instructions’ result
    - Key observation: instead of shifting the divisor to the right in each iteration, the same effect can be achieved by shifting the dividend to the left

Division algorithm (A shifted left)

- Execution of the usual algorithm (iteration 1)

A,Q = 1001010 Q = 0

B′ = 1000000

0001010 Q = 1

0010101 ← new A,Q

Division algorithm (A shifted left)

- Execution of the usual algorithm (iteration 2)

A,Q = 0010101 Q = 1

B′ = 1000000

No subtraction

0101010 ← new A,Q Q = 10

Division algorithm (A shifted left)

- Execution of the usual algorithm (iteration 3)

A,Q = 0101010 Q = 10

B′ = 1000000

No subtraction

1010100 ← new A,Q Q = 100

Division algorithm (A shifted left)

- Execution of the usual algorithm (iteration 4)

A,Q = 1010100 Q = 100

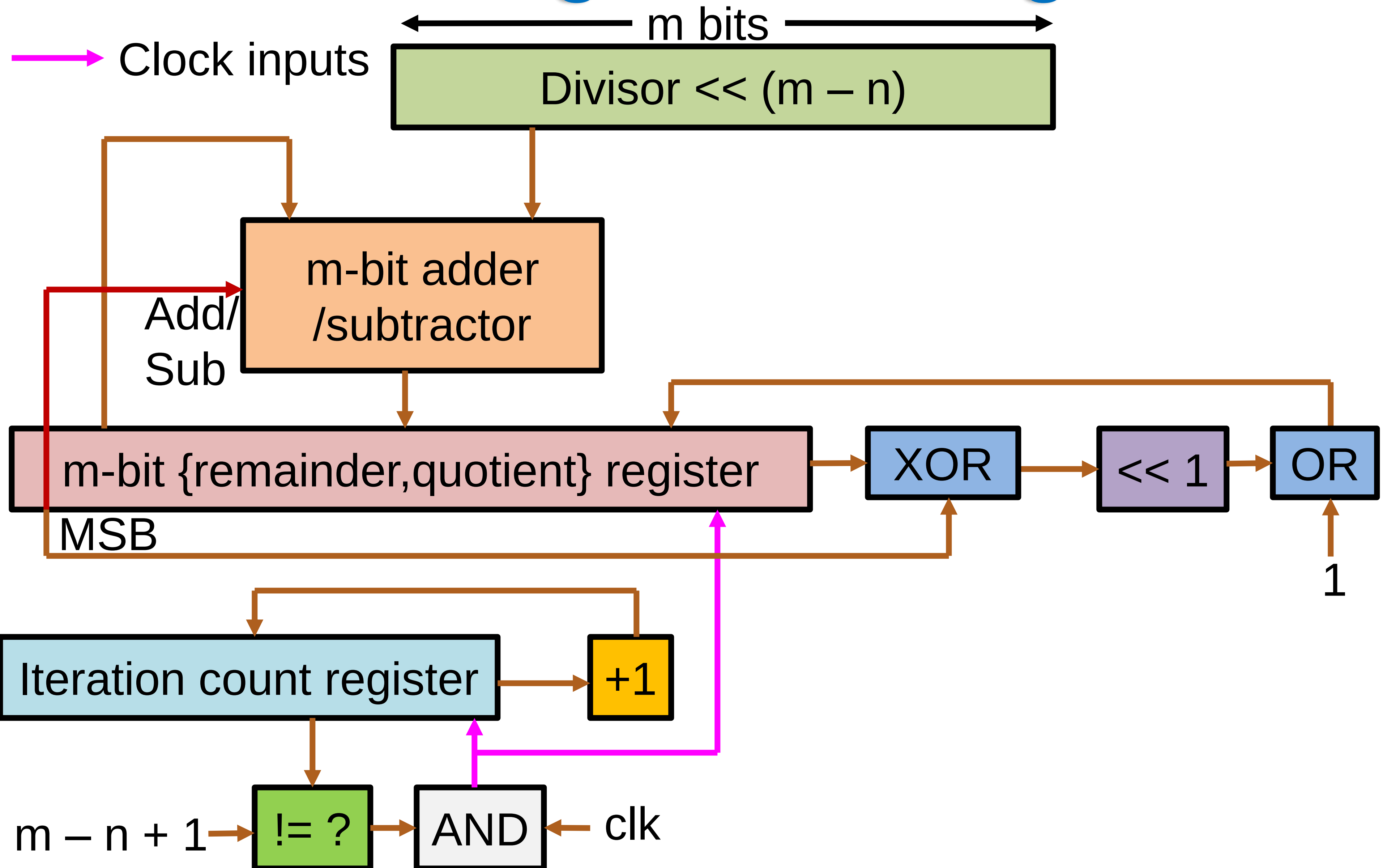
B′ = 1000000

0010100 Q = 1001

Remainder Quotient = 1001

  - Done executing four iterations (m – n + 1 iterations)
  - The remainder appears in upper 4 bits (n=4) of residual {A,Q} and quotient is the concatenation of the lower 3 bits (m – n = 3) of residual {A, Q} and the last bit of quotient

Non-restoring division algorithm



Signed division

- Suppose A and B are positive and we have $A = BQ + R$
- So, $-A = -BQ - R$ is the expected result of dividing $-A$ by B
- Rule: remainder must have the same sign as the dividend

Signed division

- Signed division algorithm
  - Convert the negative operand(s) to their positive value(s) and use any unsigned division algorithm
  - If both operands were positive originally, stop
    - $A = BQ + R$
  - If both operands were negative originally, negate remainder
    - $-A = -BQ + (-R)$
  - If dividend was negative and divisor was positive, negate both quotient and remainder
    - $-A = B(-Q) + (-R)$
  - If dividend was positive and divisor was negative, negate quotient
    - $A = -B(-Q) + R$

Signed division

- MIPS has both signed and unsigned division
  - div instruction treats operands as signed
  - divu treats operands as unsigned
- Faster division algorithms are non-trivial
  - The subtractions cannot be done in parallel
  - Fast algorithms try to predict quotient bits and detect and correct mispredictions on the fly

Floating-point addition

- Assume that the operands are represented in IEEE 754 format
- Step 1: Align exponents of the operands by shifting significand of the smaller number to the right (recall significand = 1.mantissa)
- Step 2: Add significands
  - Use 2's complement integer arithmetic ignoring the binary point
  - If result is negative, convert to its positive value
  - Put back the binary point after addition and assign the sign bit depending on the result's sign
- Step 3: Normalize the sum
  - Shift right and increase exponent OR shift left and decrease exponent

Floating-point addition

- Step 4: If overflow or underflow detected, raise exception
- Step 5: Round mantissa of the normalized sum to fit the precision
  - This may make the result unnormalized e.g., 1.11111…1 in binary becomes 10.00000…0 after half-way rounding rule is applied
- If not normalized, repeat steps 3, 4, 5; else stop
- Precision could be lost in intermediate computations e.g., during shifting significand
  - To avoid this, IEEE 754 standard recommends use of three additional bits in intermediate representations (guard bit, round bit, sticky bit)

Use of guard, round, sticky bits

- The guard and round bits extend the mantissa by two bits
- The sticky bit appears after the round bit and is set to 1 if any bit to the right of the round bit is 1
- These bits improve the precision of arithmetic whenever the significand needs to be shifted to the right
  - These bits retain a few mantissa bits which would otherwise get dropped
- If the normalization step needs to shift the significand to the left, these bits will start contributing to the precision of the mantissa

Floating-point addition

- Example: 0.5 + (-0.4375)
  - $0.5 = 1.000 \times 2^{-1}$, $-0.4375 = -1.110 \times 2^{-2}$
- Step 1: Align exponents
  - $-1.110 \times 2^{-2} = -0.111 \times 2^{-1}$
- Step 2: Add significands
  - 1.000 + (-0.111)
  - 1000 + 1001 = 0001 (2' complement)
  - 0.001 after putting back the binary point
- Step 3: Normalize
  - $0.001 \times 2^{-1} = 1.000 \times 2^{-4} = 0.0625$
  - No overflow or underflow because exponent is within legitimate range; no need to round also

Floating-point multiplication

- Step 1: Add the biased exponents of the operands and subtract the bias to get the new biased exponent
- Step 2: Multiply significands
  - Use unsigned integer multiplication algorithm ignoring the binary point
  - Put the binary point after N bit positions from right where N is the sum of the lengths of the mantissa of two operands
- Step 3: Normalize the product
- Step 4: If overflow or underflow detected, raise exception
- Step 5: Round the product; if product is not normalized, repeat steps 3, 4, 5

Floating-point multiplication

- Step 6: If the signs of the operands are same, the product is assigned positive sign; otherwise the product is negative
- Example: 0.5 x (-0.4375)
  - $(1.000 \times 2^{-1}) \times (-1.110 \times 2^{-2})$
  - New biased exponent = (127 – 1) + (127 – 2) – 127 = 124 = (127 – 3)
    - Actual exponent -3
  - Multiply significands: 1000 x 1110 = 1110000
    - Put back binary point: 1.110000
  - Unsigned product = 1.110×2^{-3}
    - Already normalized and no overflow
    - No need to round
  - Final product = -1.110×2^{-3} = -0.21875