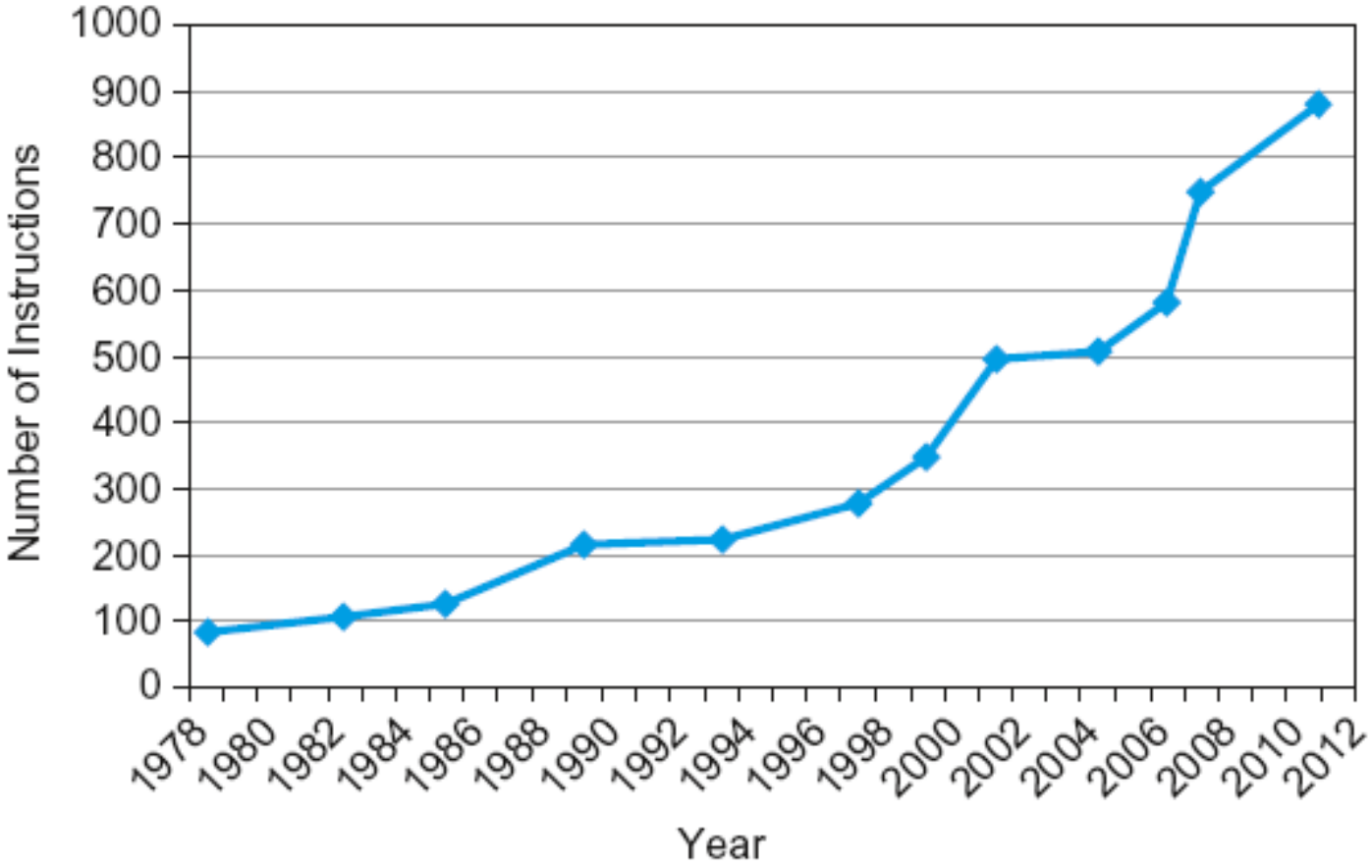
MIPS vs ARM vs INTEL

- ISA Design approach: RISC (MIPS & ARM) vs CISC (Intel)
  - CISC: single instruction can do multiple things
  - Impact instruction encoding
- Power vs performance
  - MIPS/ARM tends to be used more in power critical (or embedded) scenarios
- Key Instruction differences are around
  - the way memory is addressed,
  - branches are executed
  - exceptions are handled,

X86 ISA Evolution

- Backward compatibility ⇒ instruction set doesn't change
  - But they do have more instructions



x86 instruction set

Arithmetic for Computers

Dr. Ahmed Zahran

WGB 182

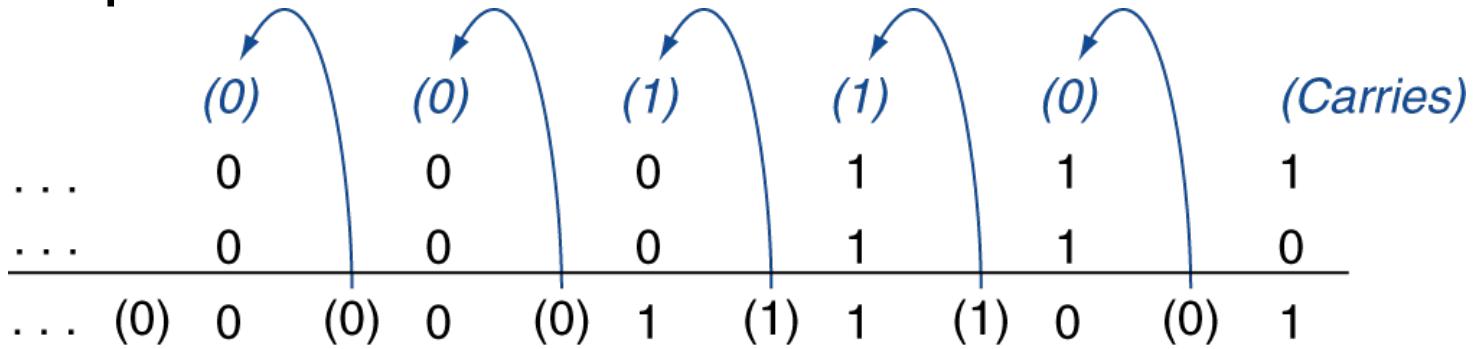
a.zahran@cs.ucc.ie

Objectives

- To understand floating number representation
- To understand how does hardware really multiply or divide numbers
- To identify how large numbers are accommodated

Integer Addition

- Example: 7 + 6



Overflow: the result is a numeric value that is outside of the range that can be represented with a given number of bits

- ***Overflow*** if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

  +7: 0000 0000 … 0000 0111
  –6: 1111 1111 … 1111 1010
  +1: 0000 0000 … 0000 0001

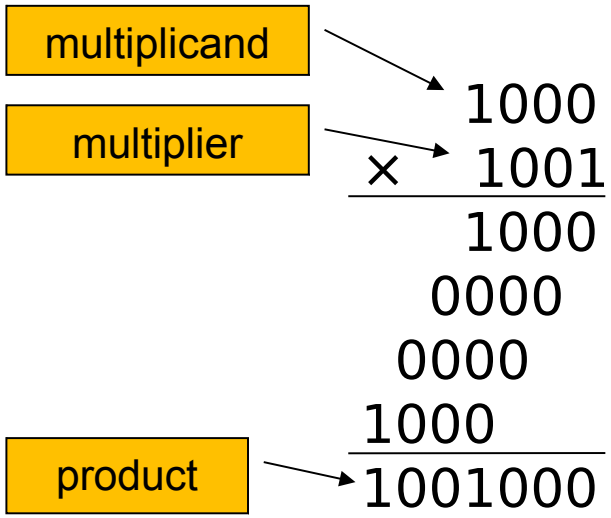
- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - Use MIPS addu, addui, subu instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS add, addi, sub instructions
  - On overflow, invoke ***exception handler***
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - mfc0 (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

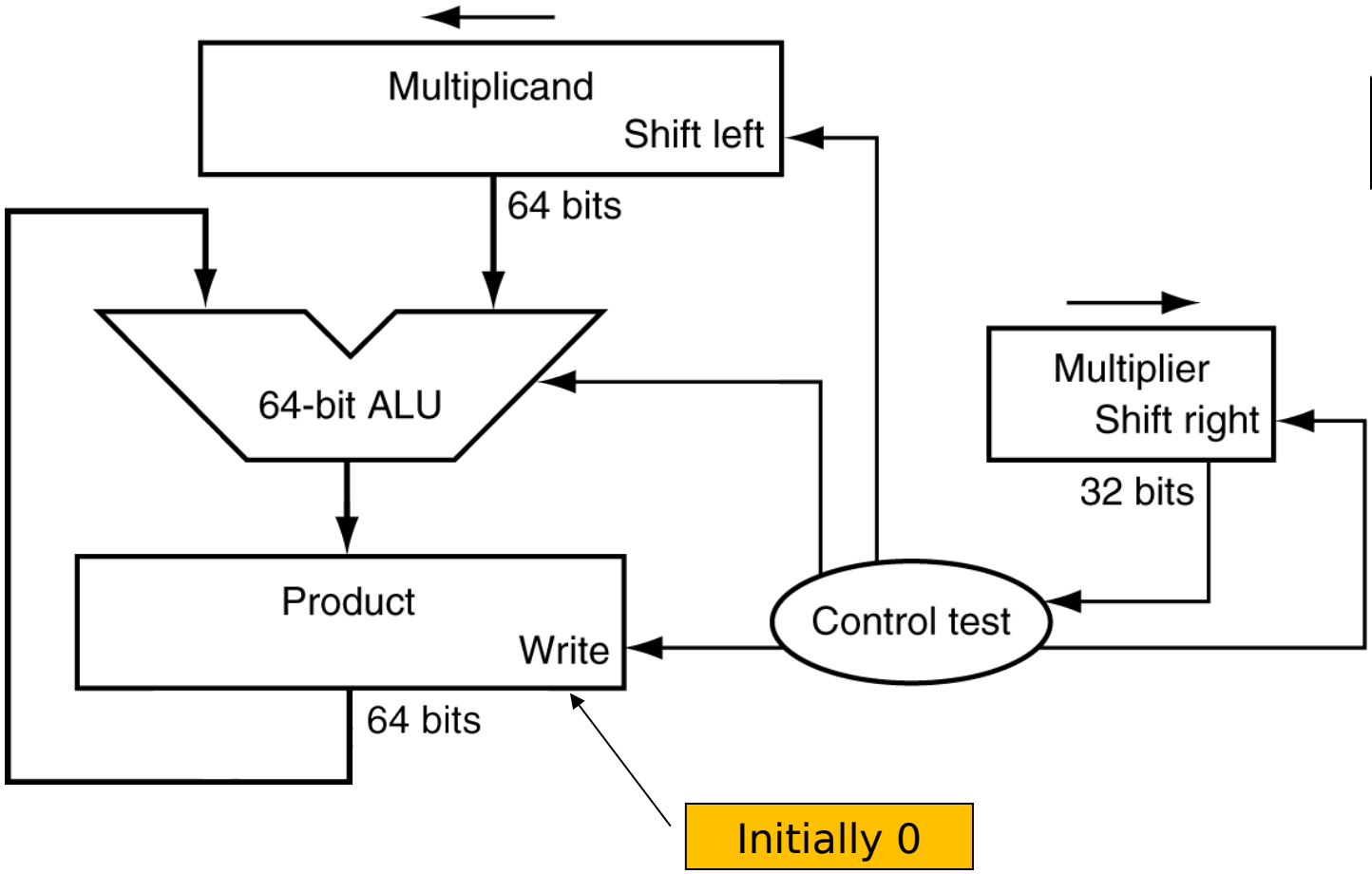
Multiplication

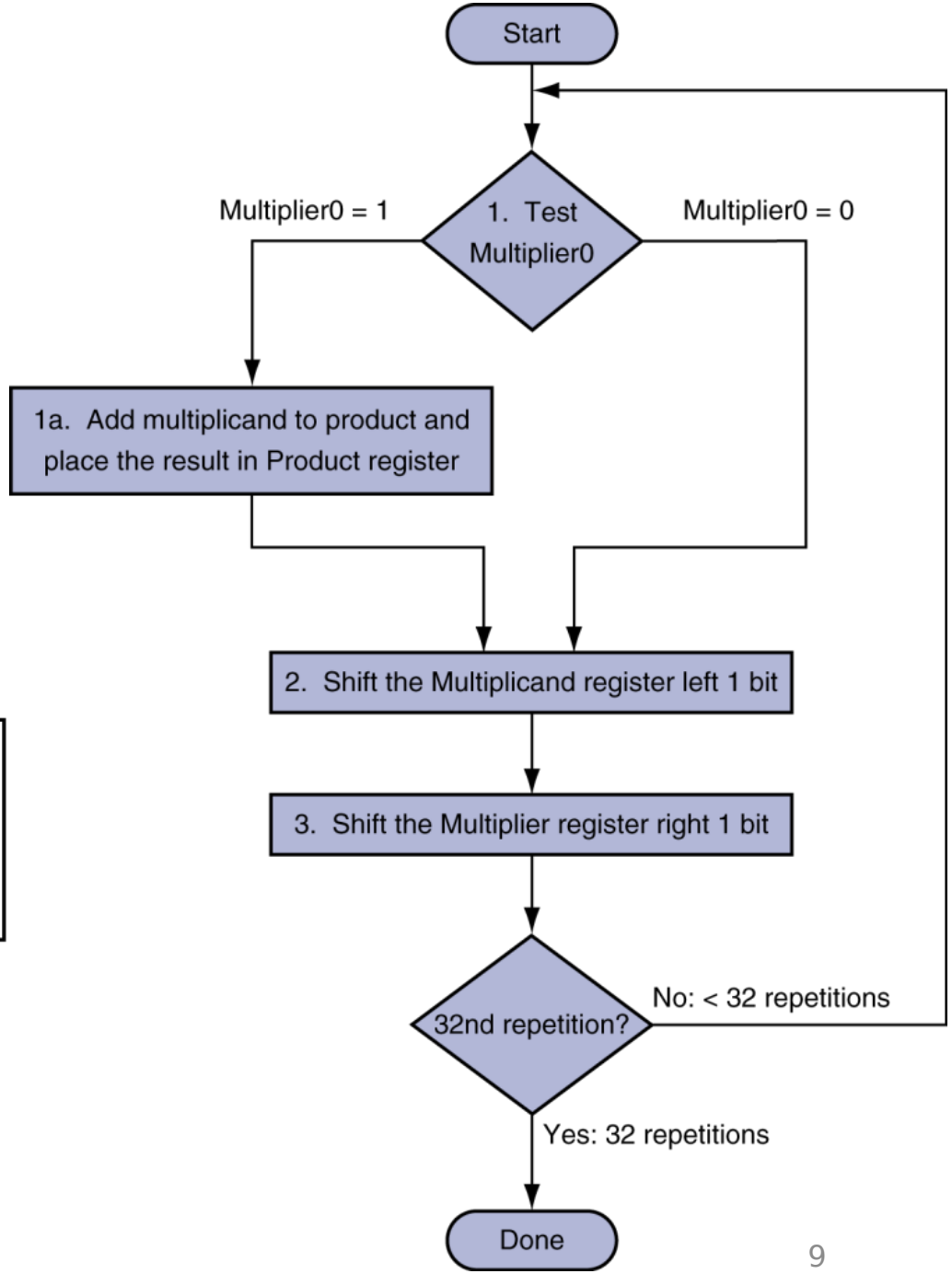
- Example: Multiply 1000 by 1001
- Start with long-multiplication approach



Length of product is the sum of operand lengths

Multiplication Hardware

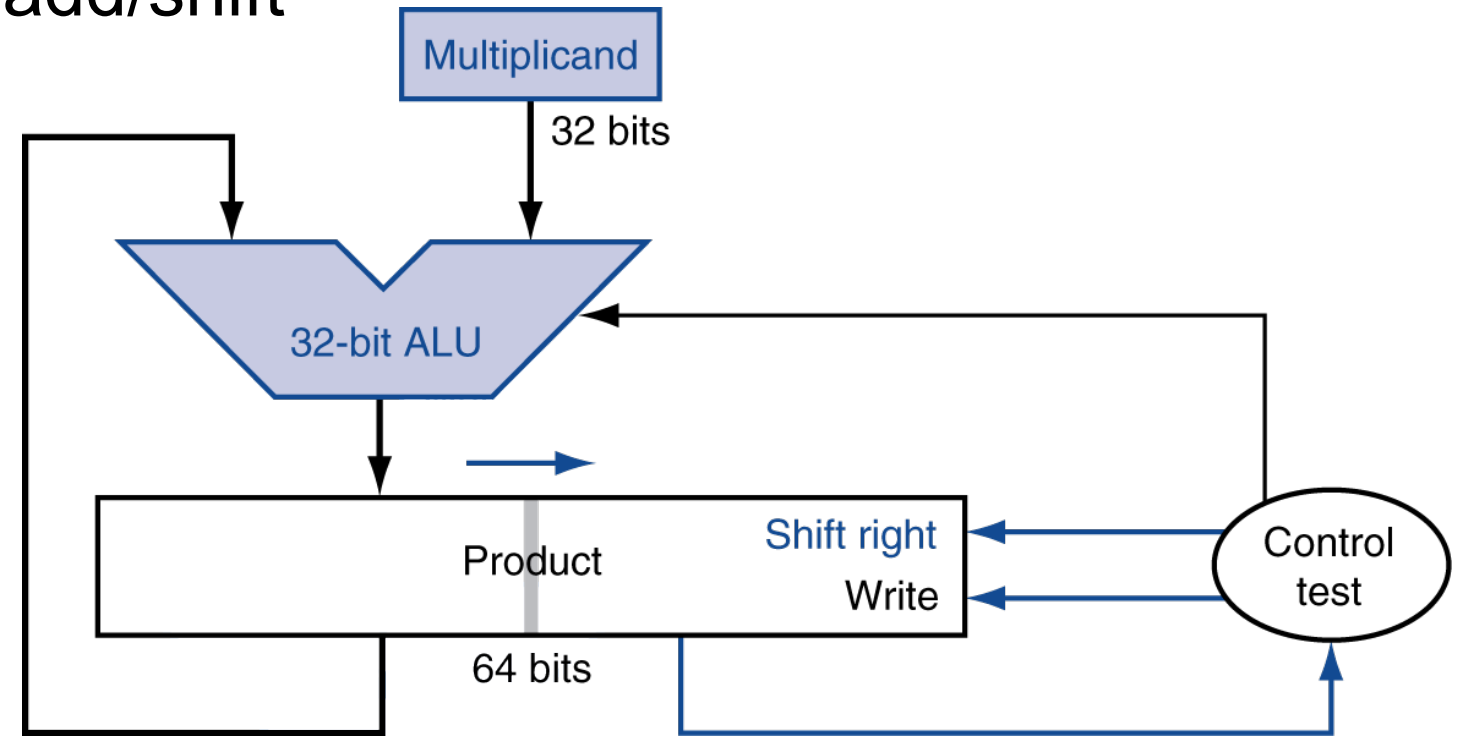




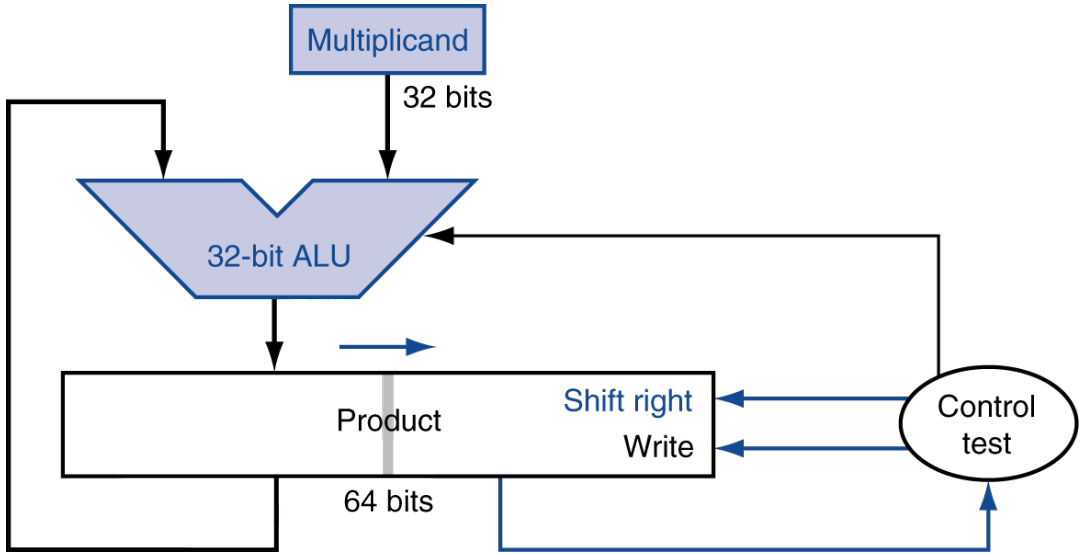
Optimised Multiplier

- Perform steps in parallel: add/shift

- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low



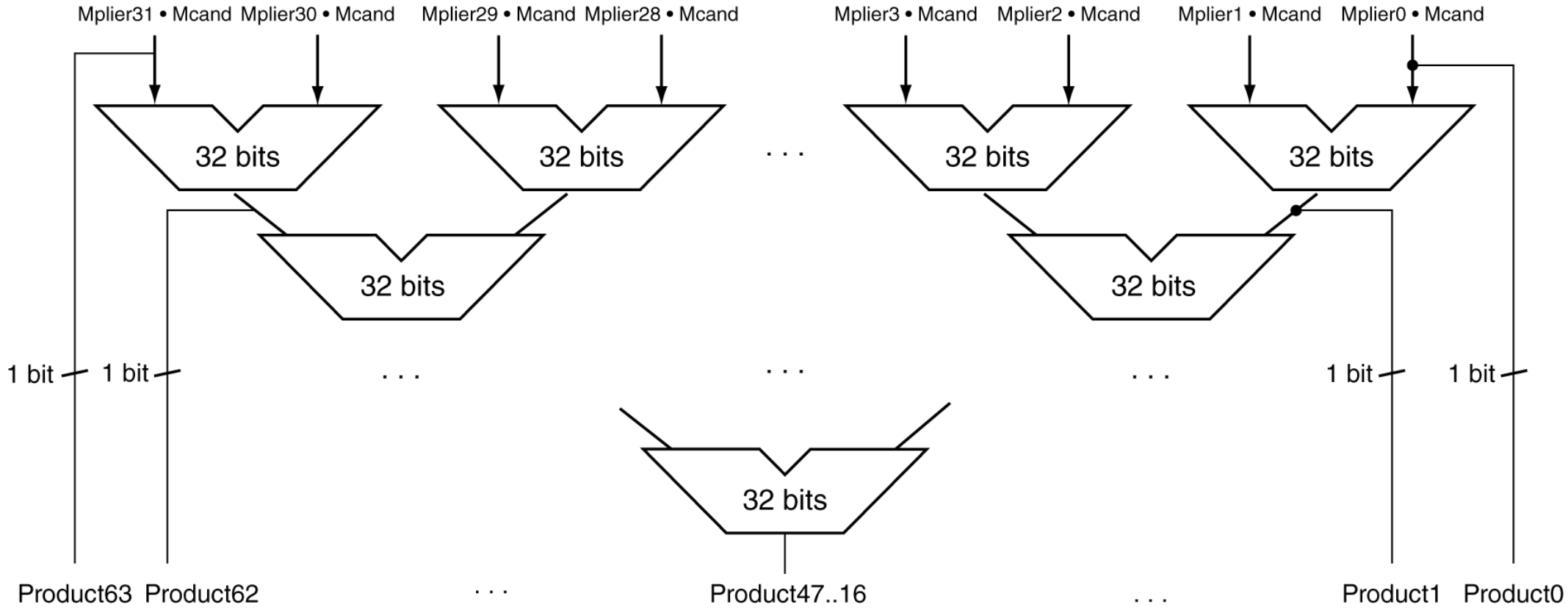
Show the register content for multiplying 6 by 4
Multiplicand remains fixed at 4 = 0100



	Product	Notes
init	0000|0110	LSB=0 → shift right
1	0100|0011	LSB=1 → addition then shift right
2	0110|0001	LSB=1 → addition then shift right
3	0011|0000	LSB=0 → shift right
4	0001|1000	Result = 24

Faster Multiplier

- Uses multiple adders [***Cost/performance tradeoff***]

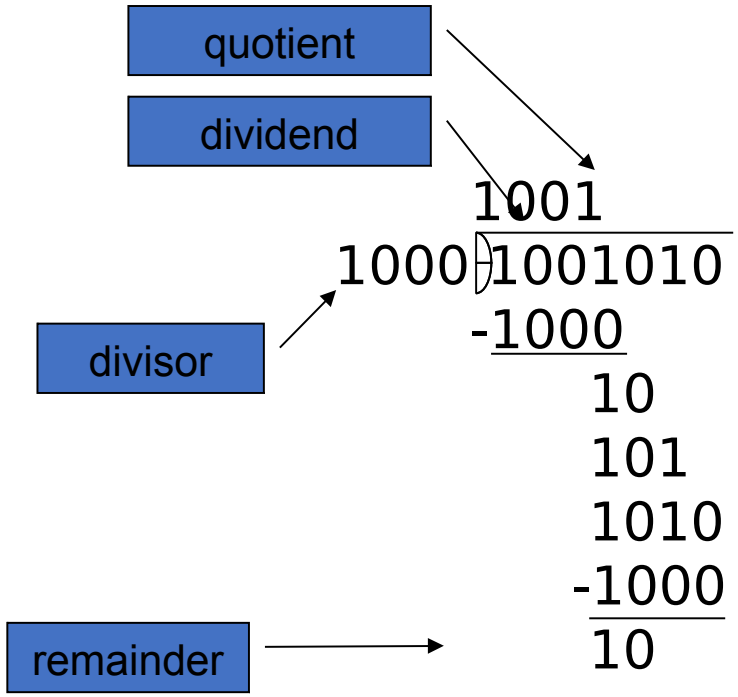


- Can be pipelined
  - Several multiplication performed in parallel

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - mult rs, rt / multu rs, rt
    - 64-bit product in HI/LO
  - mfhi rd / mflo rd
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits

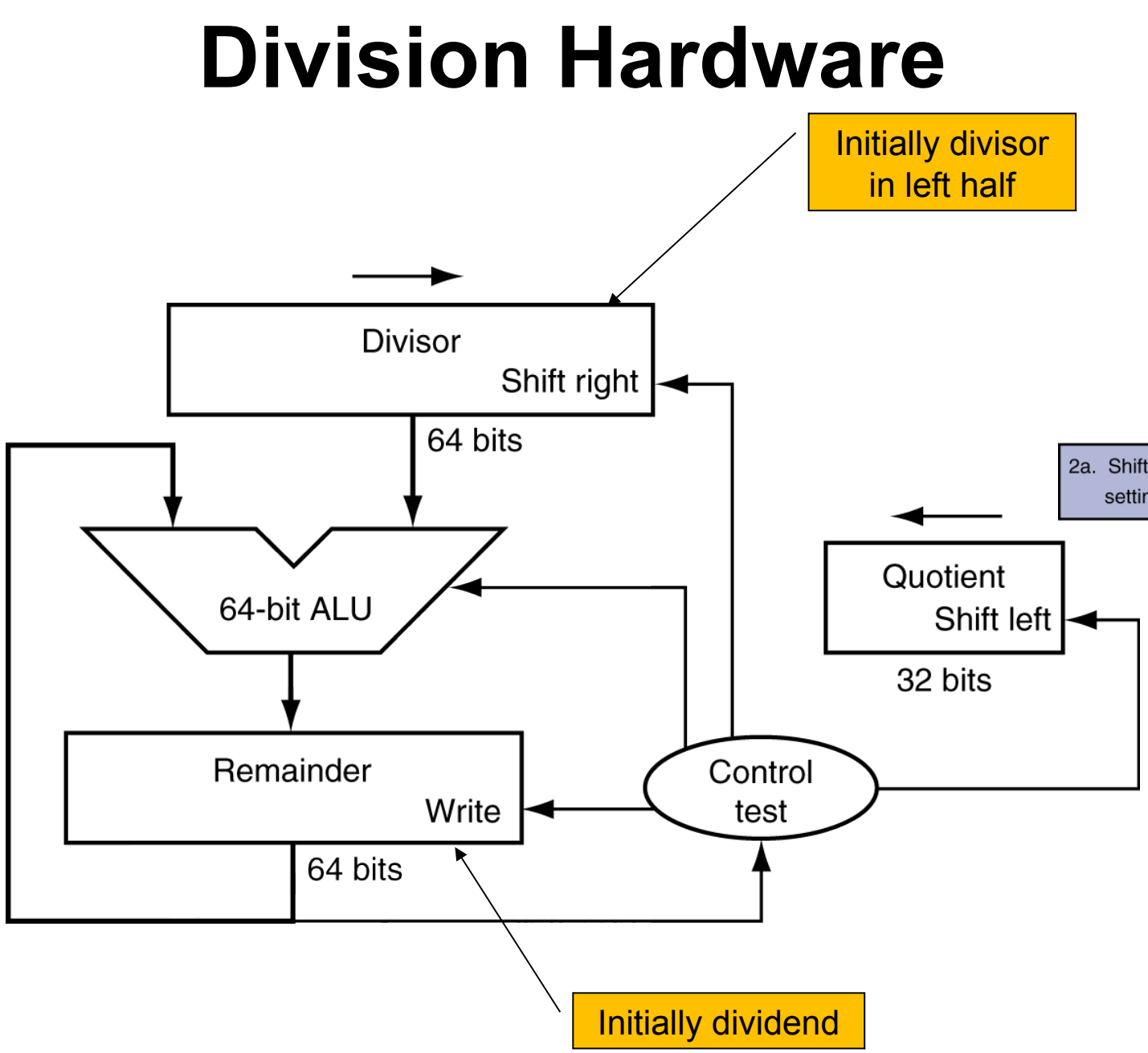
Division

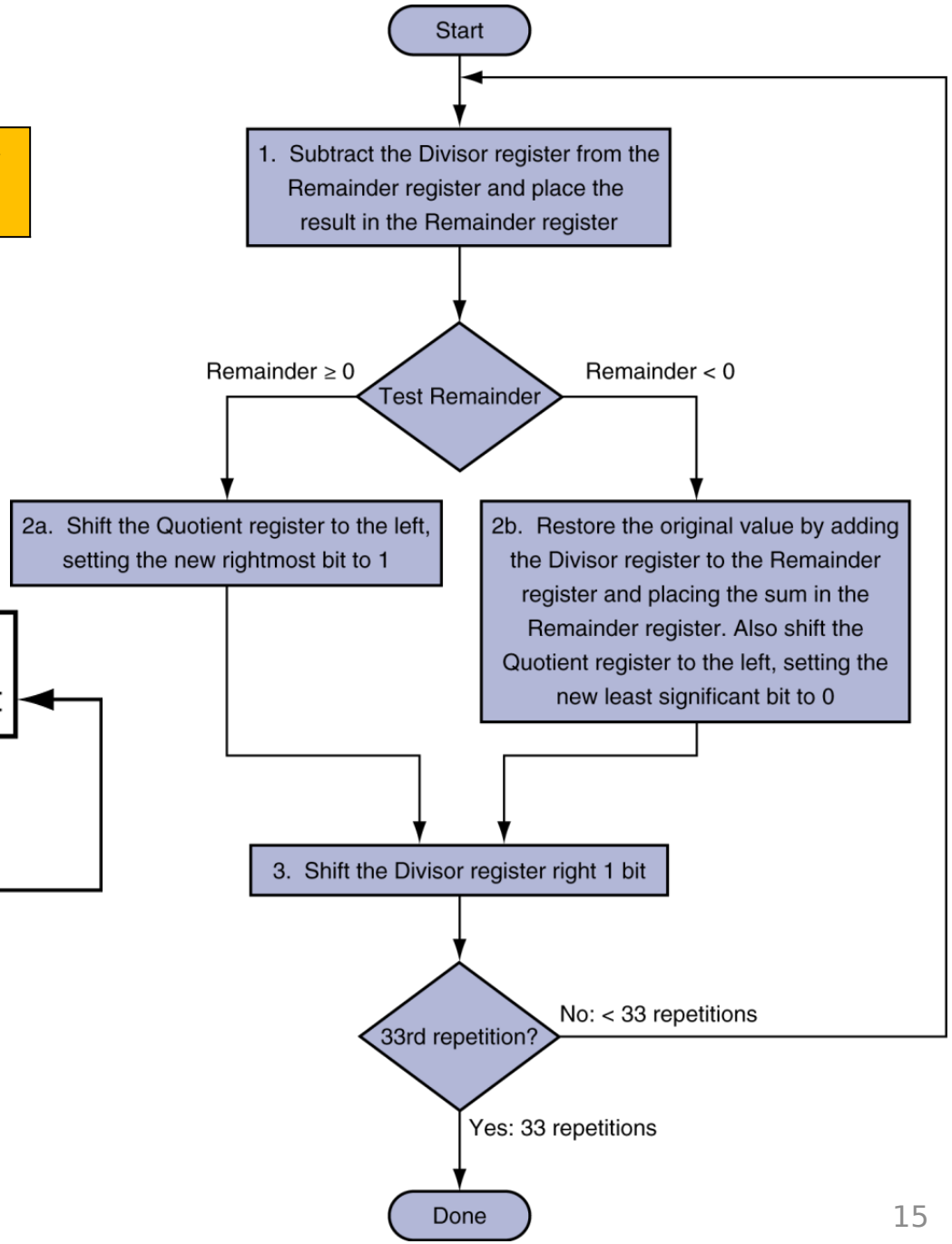


n-bit operands yield *n*-bit quotient and remainder

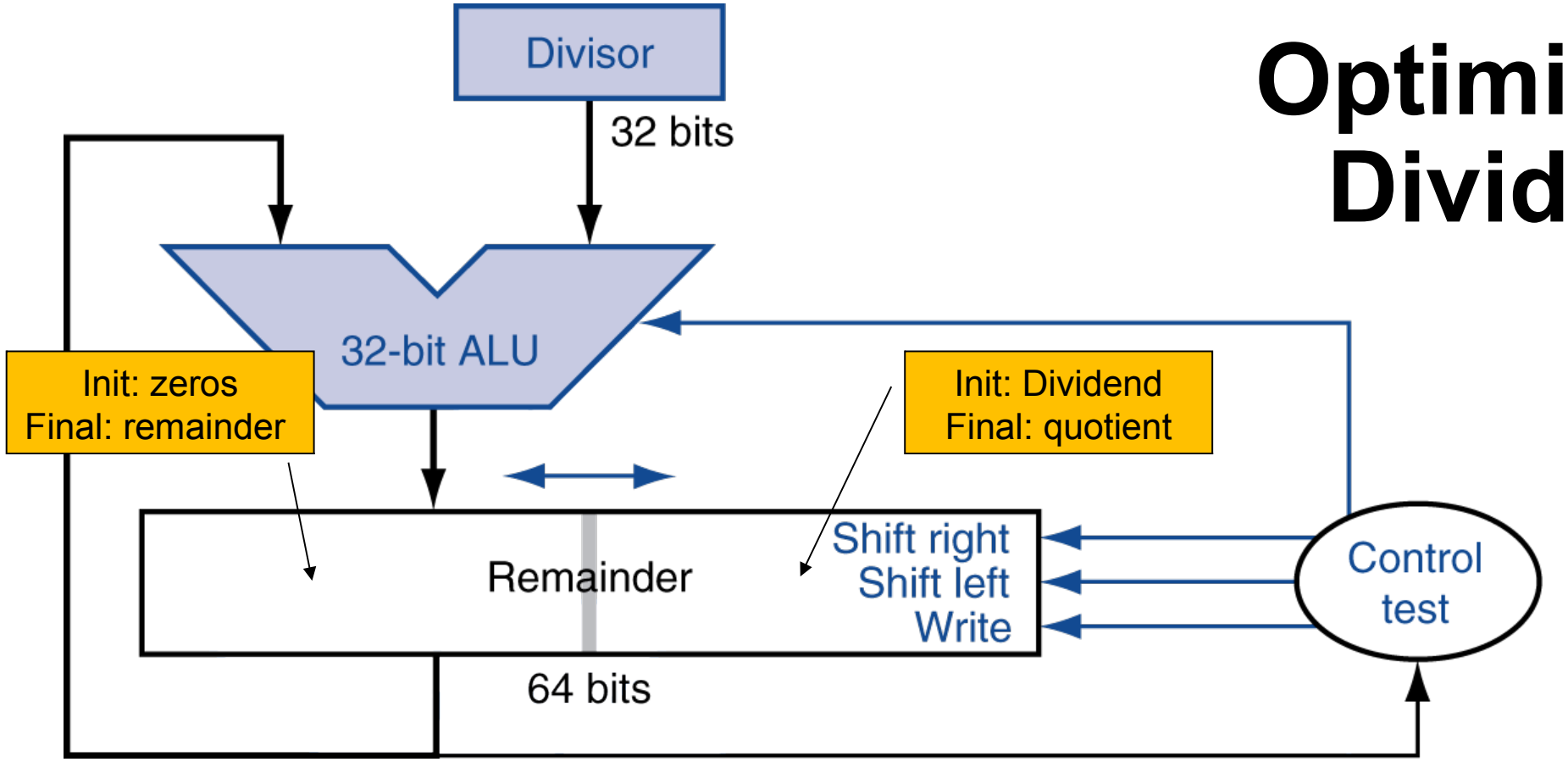
- Example divide 1001010_2 by 1000_2
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Division Hardware





Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers generate multiple quotient bits per step
  - Still require multiple steps
  - Solution currently uses future prediction and correction strategy

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - div rs, rt / divu rs, rt
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use mfhi, mflo to access result

Reading

- Sections 3.1 – 3.4