

182.690 RECHNERSTRUKTUREN – ARITHMETIC

Thomas Polzer
tpolzer@ecs.tuwien.ac.at
Institut für Technische Informatik

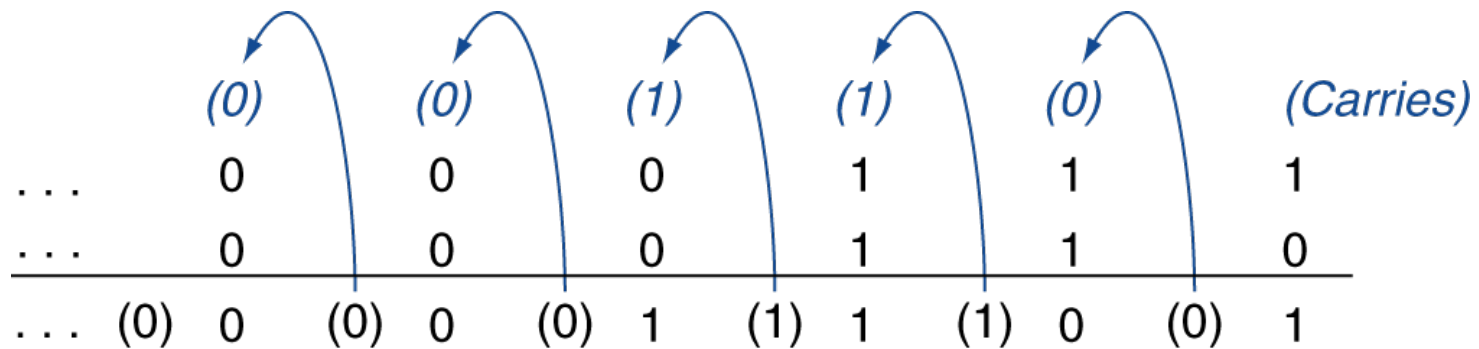


Arithmetic for Computers

- Operations on integers
 - Addition and subtraction
 - Multiplication and division
 - Dealing with overflow
- Floating-point real numbers

Integer Addition

■ Example: $7 + 6$



■ Overflow if result out of range

- Adding a positive and a negative operands → No overflow
- Adding two positive operands → Overflow if result sign is 1
- Adding two negative operands → Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: $7 - 6$

$$7 - 6 = 7 + (-6)$$

+7: 0000 0000 ... 0000 0111

-6: 1111 1111 ... 1111 1010

+1: 0000 0000 ... 0000 0001

- Overflow if result out of range
 - Subtracting two positive or two negative operands → No overflow
 - Subtracting positive from negative operand → Overflow if result sign is 0
 - Subtracting negative from positive operand → Overflow if result sign is 1

Dealing with Overflow (1)

- Overflow: Result cannot be represented in 32-bits

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

- MIPS: Overflow raises exception (aka interrupt)

Dealing with Overflow (2)

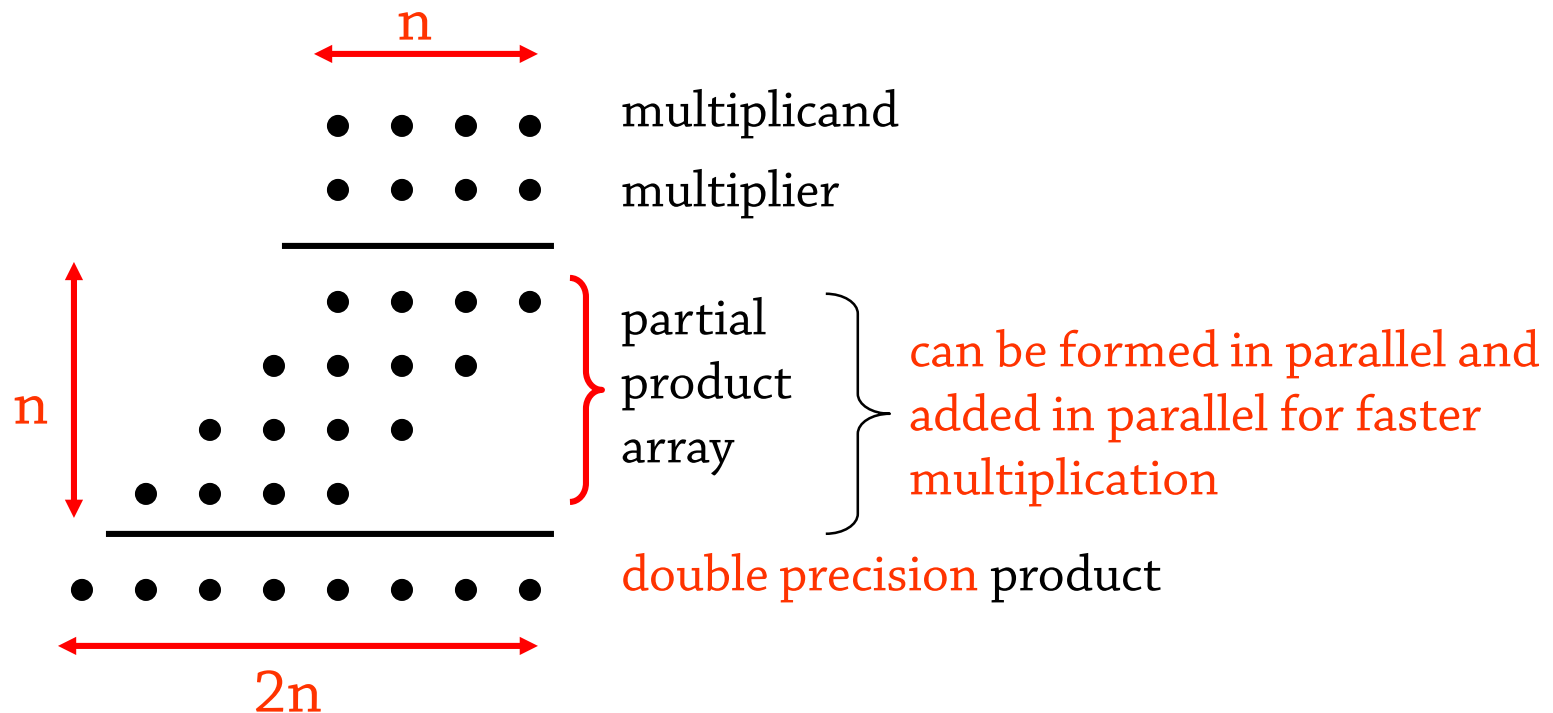
- Some languages (e.g., C) ignore overflow
 - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
 - Use MIPS `add`, `addi`, `sub` instructions
 - On overflow → Exception handler invoked

Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
 - Use 64-bit adder, with partitioned carry chain
 - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
 - SIMD (single-instruction, multiple-data)
- Saturating operations
 - On overflow, result is largest representable value
 - e.g., clipping in audio, saturation in video

Multiplication

- Binary multiplication is just a bunch of right shifts and adds



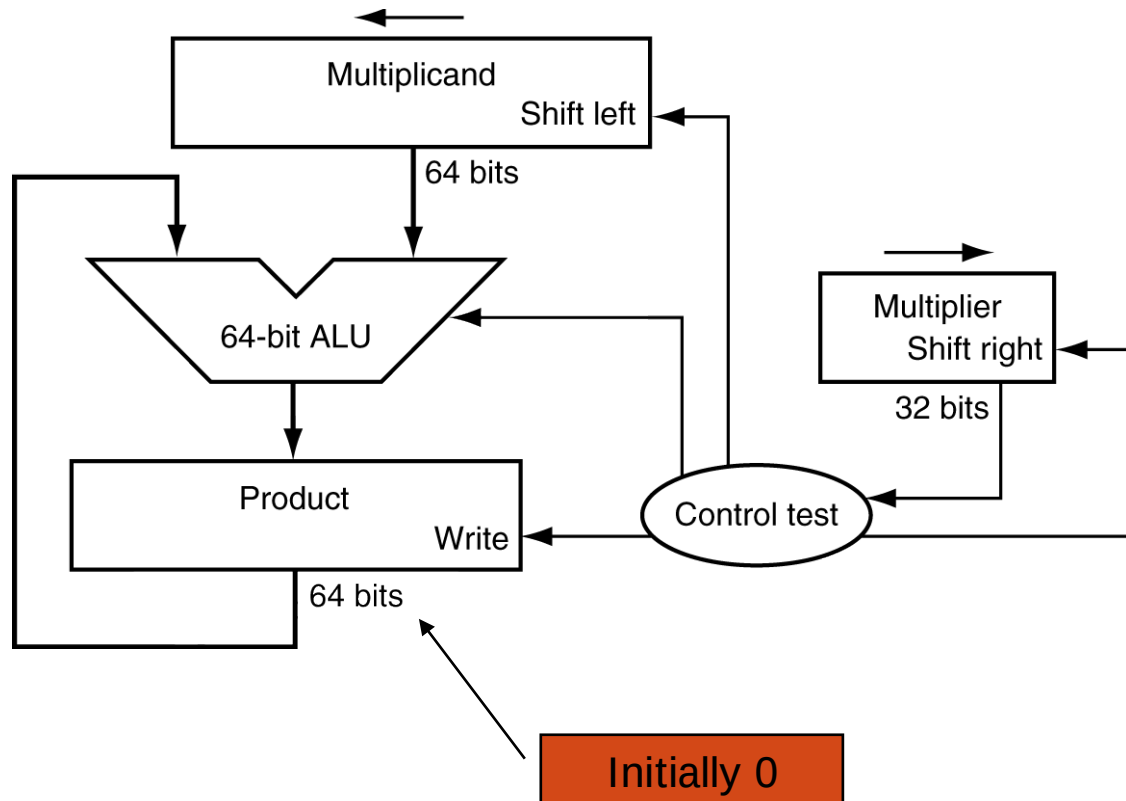
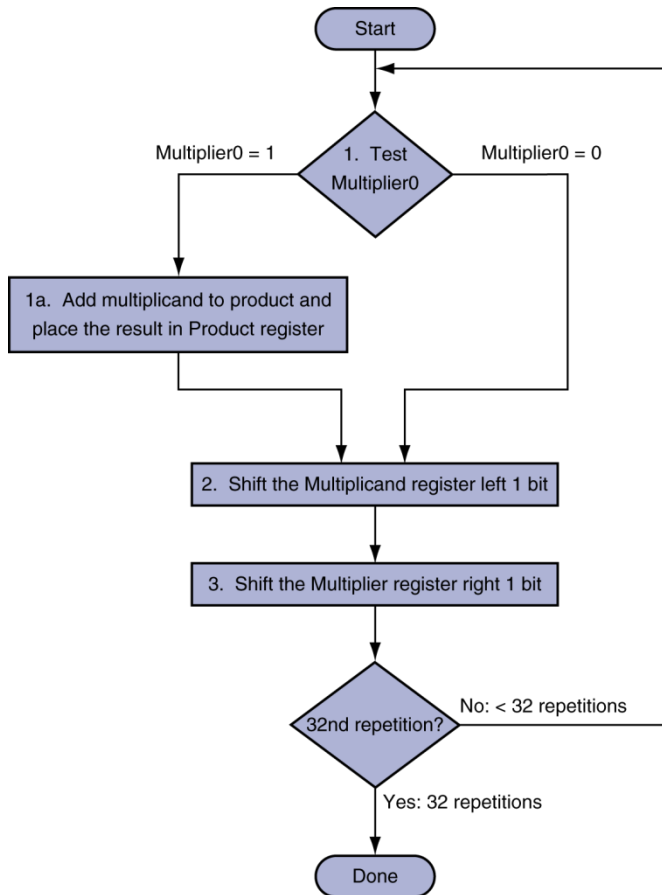
Long-multiplication Approach

- Example: $8 * 9$

$$\begin{array}{r} 1000 \\ \times 1001 \\ \hline 1000 \\ 0000 \\ 0000 \\ 1000 _ \\ \hline 1001000 = 72 \end{array}$$

- Length of product is the sum of operand lengths

Multiplication Hardware

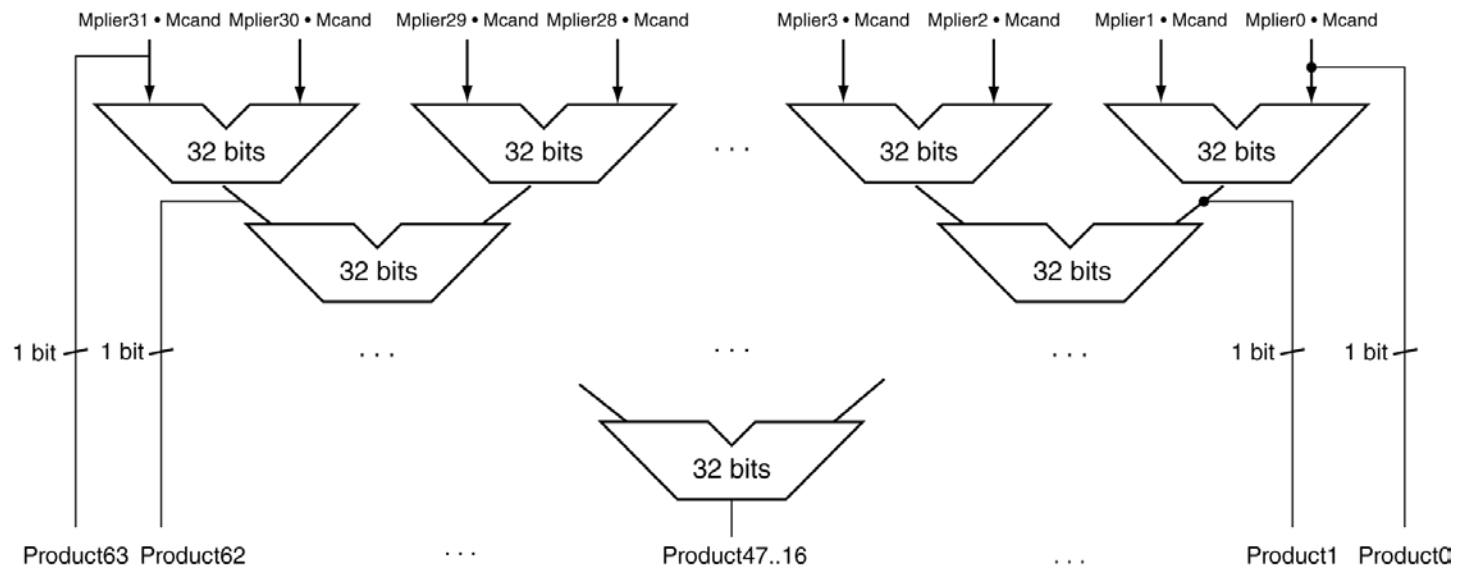


Multiplication Example (2 x 3)

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001 ¹	0000 0010	0000 0000
1	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	000 ¹	0000 0100	0000 0010
2	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000 ⁰	0000 1000	0000 0110
3	1: 0 $\Rightarrow$ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000 ⁰	0001 0000	0000 0110
4	1: 0 $\Rightarrow$ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Faster Multiplier

- Uses multiple adders
 - Cost/performance tradeoff



- Can be pipelined
 - Several multiplication performed in parallel

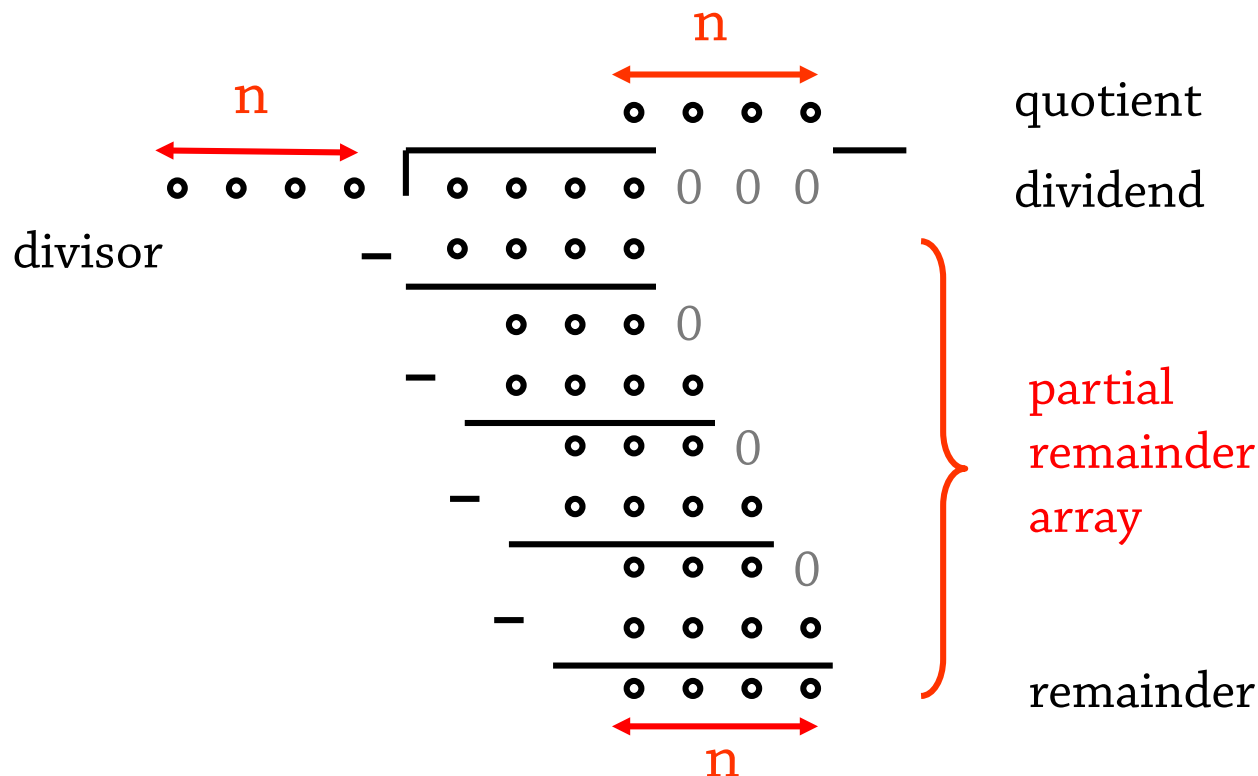
MIPS Multiplication

- 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
 - `mul rd, rs, rt`
 - Least-significant 32 bits of product → rd

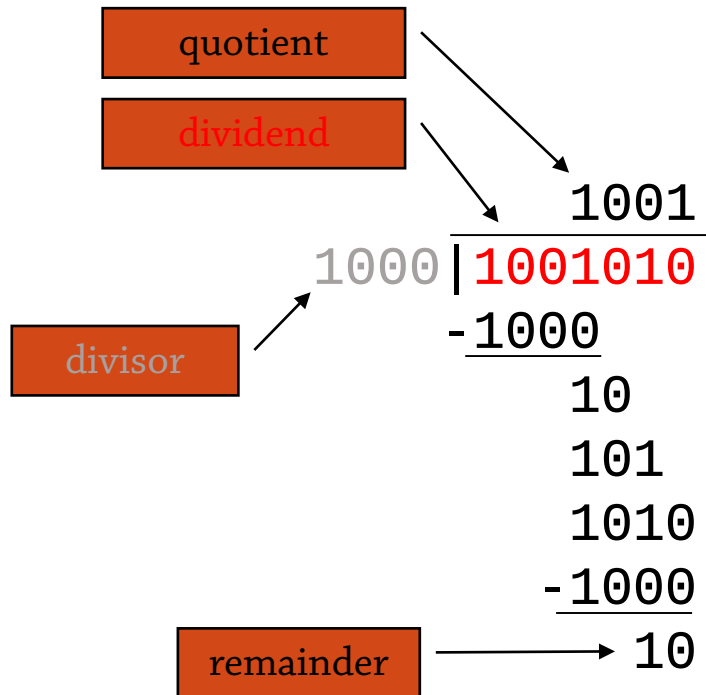
Division

- Division is just a bunch of quotient digit guesses and left shifts and subtracts

$$\text{dividend} = \text{quotient} \times \text{divisor} + \text{remainder}$$



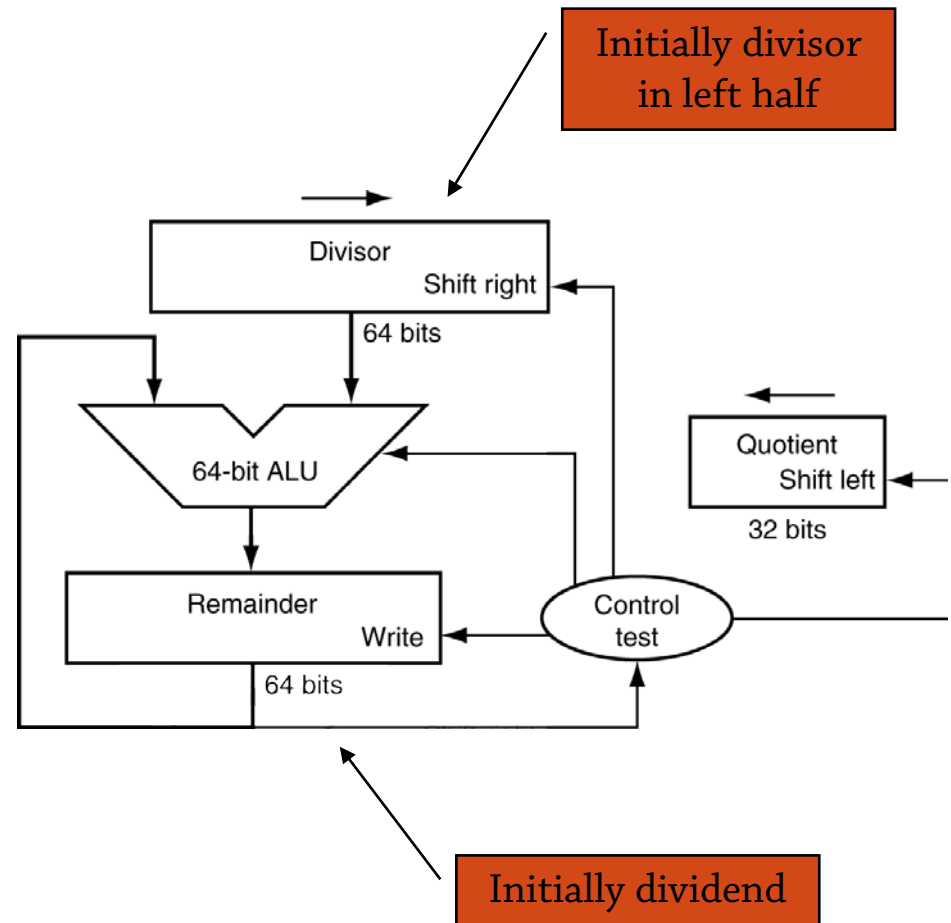
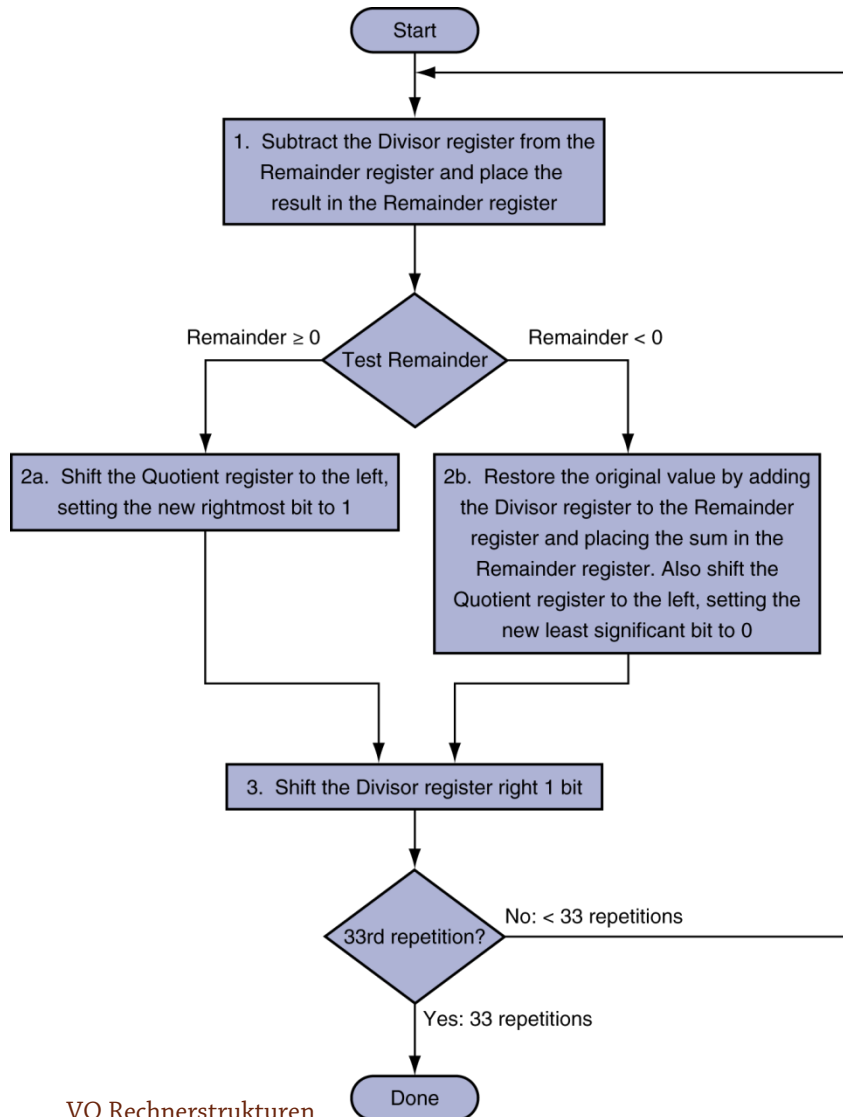
Division



n -bit operands yield n -bit quotient and remainder

- Check for 0 divisor
- Long division approach
 - If divisor $\leq$ 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



Division Example (7:2)

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	①110 0111
	2b: Rem < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem - Div	0000	0001 0000	①111 0111
	2b: Rem < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	①111 1111
	2b: Rem < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem - Div	0000	0000 0100	①000 0011
	2a: Rem $\geq$ 0 $\Rightarrow$ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem - Div	0001	0000 0010	①000 0001
	2a: Rem $\geq$ 0 $\Rightarrow$ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

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

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

Right Shift and Division

- Left shift by i places multiplies an integer by 2^i
- Right shift divides by 2^i ?
 - Only for unsigned integers
- For signed integers
 - Arithmetic right shift: replicate the sign bit
 - e.g., $-5 / 4 \rightarrow q = -1$
 - $11111011_2 \gg 2 = 11111110_2 = -2$
 - c.f. $11111011_2 \ggg 2 = 00111110_2 = +62$

Floating Point

- Representation for non-integral numbers
 - Including very small and very large numbers
- Like scientific notation
 - -2.34×10^{56} ← **normalized**
 - $+0.002 \times 10^{-4}$ ← **not normalized**
 - $+987.02 \times 10^9$ ← **not normalized**
- In binary
 - $\pm 1.xxxxxxxx \times 2^{yyyy}$
- Types float and double in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
 - Portability issues for scientific code
- Now almost universally adopted
- Two representations
 - Single precision (32-bit)
 - Double precision (64-bit)

IEEE Floating-Point Format

single: 8 bits

single: 23 bits

double: 11 bits

double: 52 bits

S	Exponent	Fraction
---	----------	----------

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
 - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
 - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + bias
 - Ensures exponent is unsigned (non-negative)
 - Single: Bias = 127; Double: Bias = 1023

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
 - Exponent: 00000001
 $\Rightarrow$ actual exponent = $1 - 127 = -126$
 - Fraction: 000...00 $\Rightarrow$ significand = 1.0
 - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
 - exponent: 11111110
 $\Rightarrow$ actual exponent = $254 - 127 = +127$
 - Fraction: 111...11 $\Rightarrow$ significand ≈ 2.0
 - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000...00 and 1111...11 reserved
- Smallest value
 - Exponent: 000000000001
 $\Rightarrow$ actual exponent = $1 - 1023 = -1022$
 - Fraction: 000...00 $\Rightarrow$ significand = 1.0
 - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
 - Exponent: 111111111110
 $\Rightarrow$ actual exponent = $2046 - 1023 = +1023$
 - Fraction: 111...11 $\Rightarrow$ significand ≈ 2.0
 - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision

- All fraction bits are significant
- Single: approx. 2^{-23}
 - Equivalent to $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
- Double: approx. 2^{-52}
 - Equivalent to $52 \times \log_{10} 2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

0.099999999403953552 < **0.1** < 0.100000000149011612

Floating-Point Example

- Represent -0.75
 - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
 - $S = 1$
 - Fraction = $1000\dots00_2$
 - Exponent = $-1 + \text{Bias}$
 - Single: $-1 + 127 = 126 = 01111110_2$
 - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: $10111111101000\dots00$
- Double: $10111111111101000\dots00$

Floating-Point Example

- What number is represented by the single-precision float

11000000101000...00

- $S = 1$
 - Fraction = $01000...00_2$
 - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 0.01_2) \times 2^{(129 - 127)}$
 $= (-1) \times 1.25 \times 2^2$
 $= -5.0$

Infinites and NaNs

- Exponent = 111...1, Fraction = 000...0
 - $\pm$ Infinity
 - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction $\neq$ 000...0
 - Not-a-Number (NaN)
 - Indicates illegal or undefined result
 - e.g., $0.0 / 0.0$
 - Can be used in subsequent calculations

IEEE 754 not only a format ...

- Rounding algorithms
- Arithmetic operations (add, subtract, multiply, divide, square root, fused-multiply-add, remainder, etc.)
- Conversions (between formats, to and from strings, etc.)
- Exception handling
Invalid ($\sqrt{-1}$), /0, over/under-flow
- <http://754r.ucbtest.org/standards/754.pdf>
(ANSI/IEEE Std 754–1985)

Floating-Point Addition

- Consider a 4-digit decimal example
 - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
 - Shift number with smaller exponent
 - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
 - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. Normalize result & check for over/underflow
 - 1.0015×10^2
- 4. Round and renormalize if necessary
 - 1.002×10^2

Floating-Point Addition

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ ($0.5 + -0.4375$)
- 1. Align binary points
 - Shift number with smaller exponent
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
 - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
 - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
 - Much longer than integer operations
 - Slower clock would penalize all instructions
- FP adder usually takes several cycles
 - Can be pipelined

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
 - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
 - Addition, subtraction, multiplication, division, reciprocal, square-root
 - $\text{FP} \leftrightarrow \text{integer}$ conversion
- Operations usually takes several cycles
 - Can be pipelined

FP Instructions in MIPS

- FP hardware is **coprocessor 1**
 - Adjunct processor that extends the ISA
- Separate FP registers
 - 32 single-precision: \$f0, \$f1, ... \$f31
 - Paired for double-precision: \$f0/\$f1, \$f2/\$f3, ...
 - Release 2 of MIPS ISA supports 32×64 -bit FP reg's
- FP instructions operate only on FP registers
 - Programs generally don't do integer ops on FP data, or vice versa
 - More registers with minimal code-size impact
- FP load and store instructions
 - **lw**, **ld**, **sw**, **sd**
 - e.g., **ld** \$f8, 32(\$sp)

FP Instructions in MIPS

- Single-precision arithmetic
 - `add.s`, `sub.s`, `mul.s`, `div.s`
 - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
 - `add.d`, `sub.d`, `mul.d`, `div.d`
 - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
 - `c.XX.s`, `c.XX.d` (`xx` is `eq`, `lt`, `le`, ...)
 - Sets or clears FP condition-code bit
 - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
 - `bc1t`, `bc1f`
 - e.g., `bc1t TargetLabel`

Limited Accuracy

- Be careful with floating point operations:

$$(1.50\text{E}+38 - 1.50\text{E}+38) + 1.0 =$$

$$= 0.0 + 1.0 = 1.0$$

$$1.50\text{E}+38 + (-1.50\text{E}+38 + 1.0) =$$

$$1.50\text{E}+38 - 1.50\text{E}+38 = 0.0$$

WHY?: Insufficient precision to add numbers with largely different exponent:

$$-1.50\text{E}+38 + 1.0 \rightarrow \text{Result is } -1.50\text{E}+38$$

Interpretation of Data

- Bits have no inherent meaning
 - Interpretation depends on the instructions applied
- Computer representations of numbers
 - Finite range and precision
 - Need to account for this in programs

182.690 RECHNERSTRUKTUREN – ARITHMETIC

Thomas Polzer
tpolzer@ecs.tuwien.ac.at
Institut für Technische Informatik

