

§3.2 Addition and Subtraction

## Integer Addition

- Example: 7 + 6

...	(0)		(0)		(1)		(1)		(0)		(Carries)	
...	0		0		0		1		1		1	
...	0		0		0		1		1		0	
...	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1
- Overflow if result out of range
  - Adding positive and negative operands, no overflow
  - Adding two positive operands
    - Overflow if result sign is 1
  - Adding two negative operands
    - Overflow if result sign is 0

Chapter 3 — Arithmetic for Computers — 3

## Integer Subtraction

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

+7:	0000	0000	...	0000	0111
-6:	1111	1111	...	1111	1010
<hr/>					
+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

Chapter 3 — Arithmetic for Computers — 4

## Dealing with Overflow

- Overflow occurs when the result of an operation cannot be represented in 32 bits
  - i.e., when the sign bit contains a value bit of the result and not the proper sign bit
  - When adding operands with different signs or when subtracting operands with the same sign, overflow can never occur

Operation	Operand A	Operand B	Result indicating overflow
$X = A + B$	$A \geq 0$	$B \geq 0$	$X < 0$
$X = A + B$	$A < 0$	$B < 0$	$X \geq 0$
$X = A - B$	$A \geq 0$	$B < 0$	$X < 0$
$X = A - B$	$A < 0$	$B \geq 0$	$X \geq 0$

Chapter 3 — Arithmetic for Computers — 5

## Ignoring 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

Chapter 3 — Arithmetic for Computers — 6

§3.3 Multiplication

# Multiplication

- Start with long-multiplication approach

multiplicand

multiplier

product

1000

x

1001

1000

0000

0000

1000

1001000

Maximum length of product is the sum of operand lengths

Multiplicand

Shift left

64 bits

64-bit ALU

Product

Write

Initially 0

Control test

Multipler

Shift right

32 bits

Chapter 3 — Arithmetic for Computers — 7

# Hardware

Start

1. Test Multiplier0

Multiplier0 = 1

Multiplier0 = 0

1a. Add multiplicand to product and place the result in Product register

2. Shift the Multiplicand register left 1 bit

3. Shift the Multiplier register right 1 bit

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done

Chapter 3 — Arithmetic for Computers — 8

e.g., 0010 x 0011

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

Chapter 3 — Arithmetic for Computers — 9

Optimized Multiplier

- Perform steps in parallel: add/shift

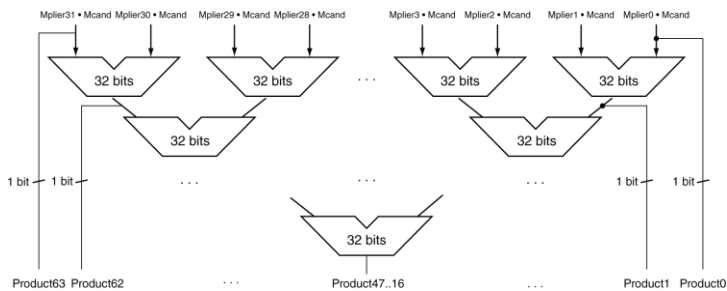
The diagram illustrates an optimized multiplier circuit. A 32-bit Multiplicand is fed into a 32-bit ALU. The ALU's output is shifted right and then written to a 64-bit Product register. A Control test unit, which receives feedback from the Product register, sends signals back to the ALU and the Shift right Write unit.

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

Chapter 3 — Arithmetic for Computers — 10

## 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

§3.4 Division

# Division

quotient

dividend

divisor

remainder

1001

1000

10

101

1010

10

1001010

-1000

-1000

10

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

Chapter 3 — Arithmetic for Computers — 13

# Division Hardware

Start

1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

Test Remainder

Remainder  $\geq 0$

2a. Shift the Quotient register to the left, setting the new rightmost bit to 1

Remainder  $< 0$

2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0

3. Shift the Divisor register right 1 bit

33rd repetition?

No:  $< 33$  repetitions

Yes: 33 repetitions

Done

Initially divisor in left half

Divisor

Shift right

64 bits

64-bit ALU

Remainder

Write

64 bits

Control test

Quotient Shift left

32 bits

Initially dividend

Chapter 3 — Arithmetic for Computers — 14

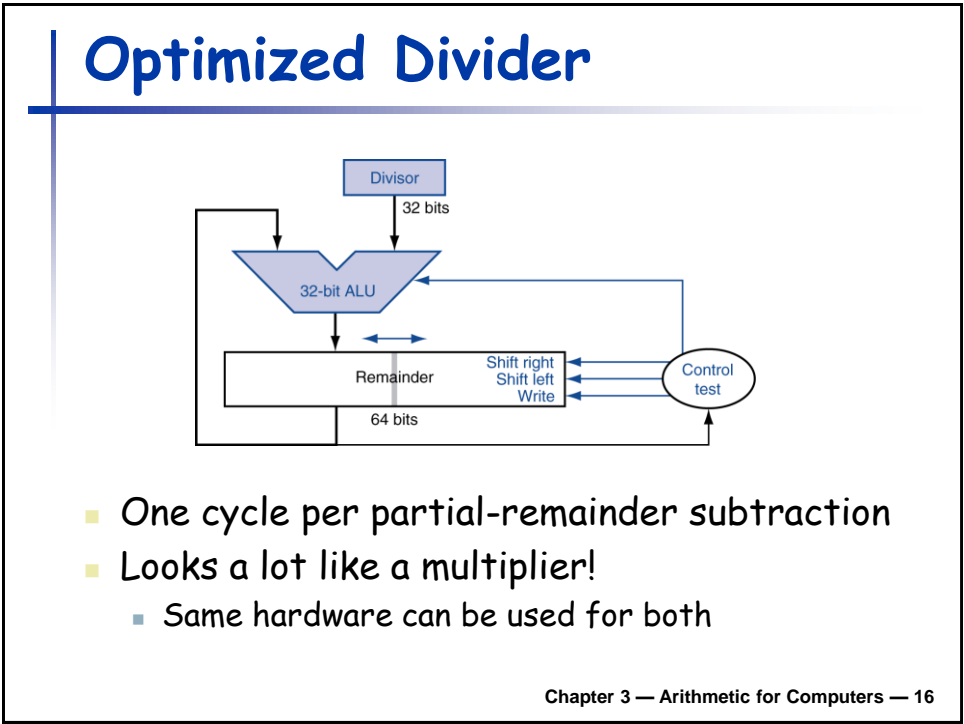
Chapter 3 — Arithmetic for Computers

7

e.g., 0111/0010

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	0110 0111
	2b: Rem < 0 ⇒ +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	0111 0111
	2b: Rem < 0 ⇒ +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	0111 1111
	2b: Rem < 0 ⇒ +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	0000 0011
	2a: Rem ≥ 0 ⇒ 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	0000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Chapter 3 — Arithmetic for Computers — 15





## Faster Division

- Can't use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g., *SRT division*) generate multiple quotient bits per step
  - Still require multiple steps
  - Also requires table lookups...

Chapter 3 — Arithmetic for Computers — 17

## 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

Chapter 3 — Arithmetic for Computers — 18

§3.9 Fallacies and Pitfalls

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 shift right: replicate sign bit
  - e.g.,  $-5 / 4$ 
    - $1111011_2 \gg 2 = 1111110_2 = -2$
    - Rounds toward  $-\infty$
  - c.f.  $1111011_2 \ggg 2 = 0011110_2 = +6$

Chapter 3 — Arithmetic for Computers — 19

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		$(x+y)+z$	$x+(y+z)$
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism

Chapter 3 — Arithmetic for Computers — 20

## Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

Chapter 3 — Arithmetic for Computers — 21

## Combinational vs. Sequential

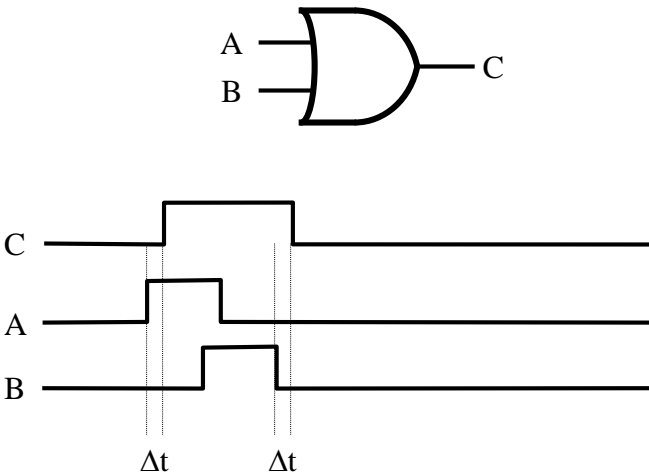
- Combinational: output depends completely on the value of the inputs
  - time doesn't matter
- Sequential: output also depends on the *state a little while ago*
  - can depend on the value of the output some time in the past
  - we need a clock for synchronization/control

Arithmetic for Computers — 22

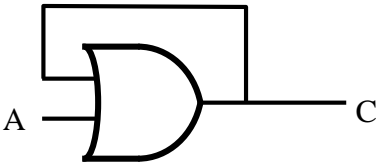
## Memory

- Think about how you might design a combinational circuit that could be used as a single bit of *memory*
- Recall that the output of a gate can change whenever the inputs change

## Gate Timing



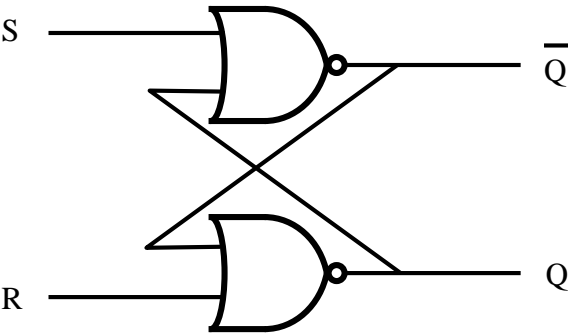
## Feedback



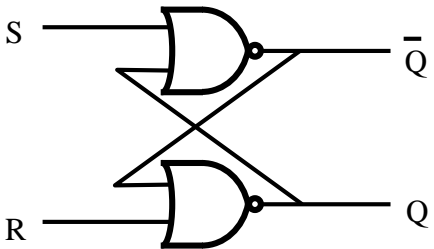
- What happens when *A* changes from 1 to 0?

## Set-Reset (S-R) latch

- Two NOR gates



### S-R latch Truth Table

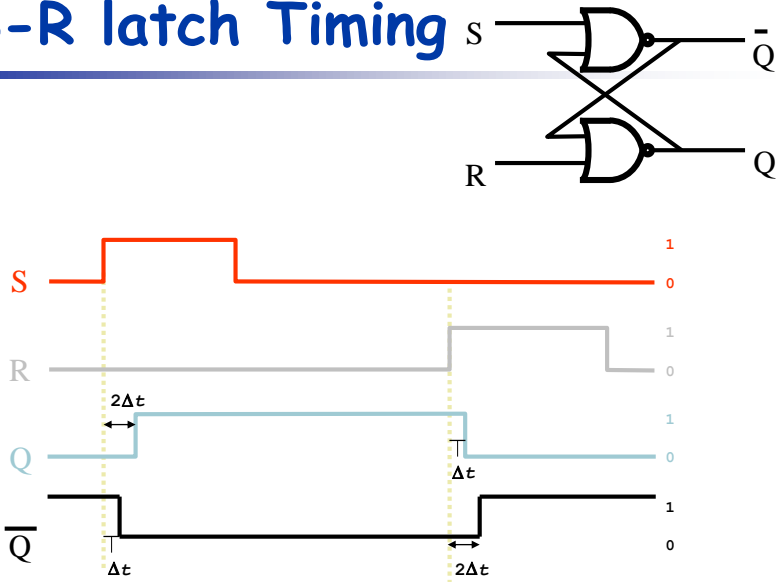


$Q_t$	$S_t$	$R_t$	$Q_{t+1}$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0?
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0?

If both  $S = 1$  and  $R = 1$ , then  $Q$ 's output is undefined

Arithmetic for Computers — 27

### S-R latch Timing



Signal	Initial	Final
S	0	0
R	0	0
Q	0	0
Q-bar	1	1

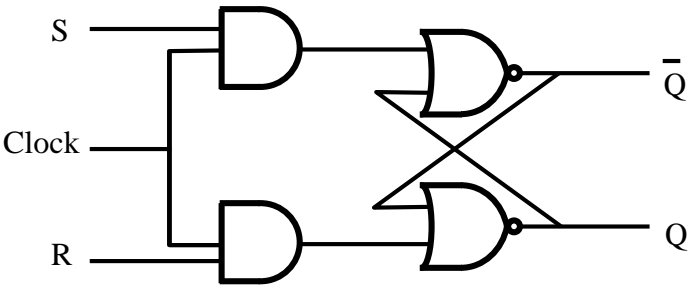
Arithmetic for Computers — 28

## Clocked S-R Latch

- Inside a computer we want the output of gates to change only at specific times
  - We can add some circuitry to make sure that changes occur only when a *clock* changes
  - i.e., when the clock changes from 0 to 1

Arithmetic for Computers — 29

## Clocked S-R Latch



- Q only changes when the Clock is a 1
- If Clock is 0, neither S nor R are able to actually *reach* the NOR gates

Arithmetic for Computers — 30

## What if $S=R=1$ ?

- The truth table earlier showed a question mark when  $S$  and  $R$  both equal 1
- The value of  $Q$  is nondeterministic
  - i.e., the circuit is not *stable*
- We need to make sure that  $S$  and  $R$  both do not equal 1 - but how?

Arithmetic for Computers — 31

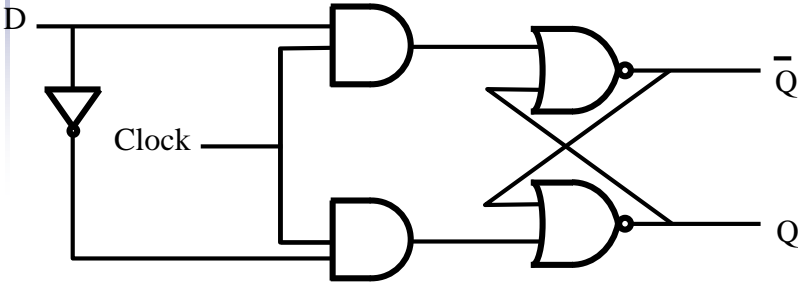
## What if $S=R=1$ ?

- The truth table earlier showed a question mark when  $S$  and  $R$  both equal 1
- The value of  $Q$  is nondeterministic
  - i.e., the circuit is not *stable*
- We need to make sure that  $S$  and  $R$  both do not equal 1 - but how?
  - Still use the clock
  - Combine  $S$  and  $R$  together

Arithmetic for Computers — 32

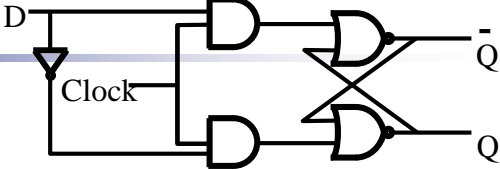


Avoiding  $S=R=1$ : D Flip-Flop



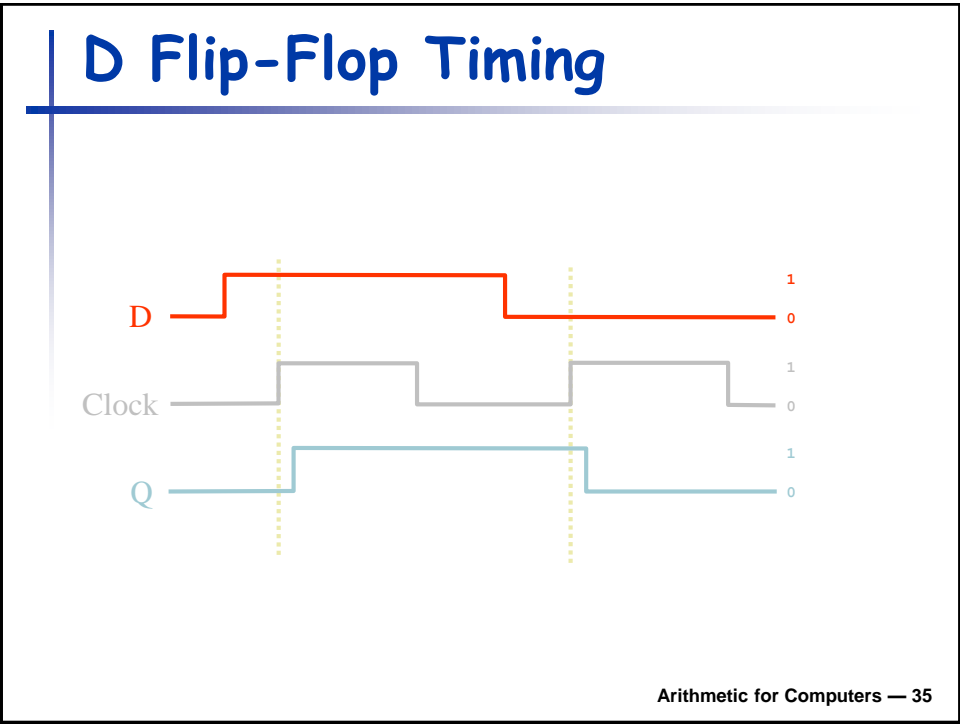
Arithmetic for Computers — 33

D Flip-Flop



- Now we have only one input: D
- If D is a 1 when the clock becomes 1, the circuit will *remember* the value 1 ( $Q=1$ )
- If D is a 0 when the clock becomes 1, the circuit will *remember* the value 0 ( $Q=0$ )

Arithmetic for Computers — 34



8-Bit Memory

- We can use eight D Flip-Flops to create an 8-bit memory
- We have eight inputs that we want to *store*, all *written* at the same time
  - all eight flip-flops use the same clock
- Can use for registers

Arithmetic for Computers — 36

