Computer Architecture and Engineering
Lecture 7: ALU Design

February 8, 1995

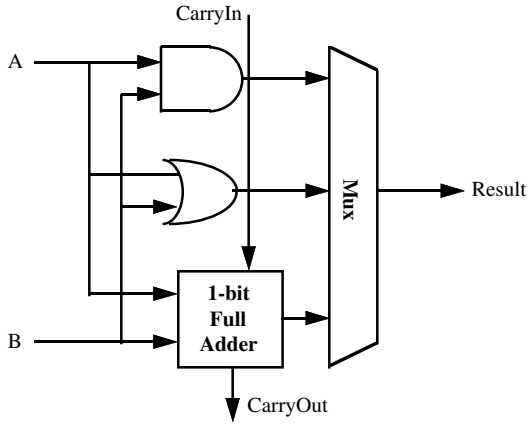
Dave Patterson (patterson@cs) and
Shing Kong (shing.kong@eng.sun.com)

Slides available on http://http.cs.berkeley.edu/~patterson

©DAP & SIK 1995

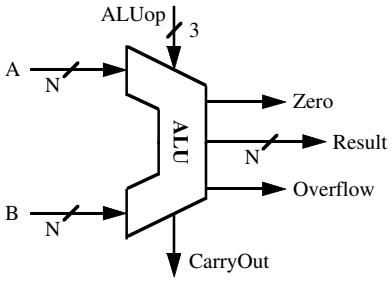
Review: A One Bit ALU

° **This 1-bit ALU will perform AND, OR, and ADD**



©DAP & SIK 1995

Review: Functional Specification of the ALU



- ALU Control Lines (ALUop) — Function
  - 000 — And
  - 001 — Or
  - 010 — Add
  - 110 — Subtract
  - 111 — Set-on-less-than

©DAP & SIK 1995

Recap of Last Lecture

- An Overview of the Design Process
  - Design is an iterative process-- successive refinement
  - Do NOT wait until you know everything before you start
- An Introduction to Binary Arithmetics
  - If you use 2's complement representation, subtract is easy.
- ALU Design
  - Designing a Simple 4-bit ALU
  - Other ALU Construction Techniques
- On-line Design Notebook
  - Open a window and keep an editor running while you work
  - Refer to the handout as an example

©DAP & SIK 1995

Outline of Today's Lecture

- Recap of Last Lecture and Introduction of Today's Lecture (2 min.)
- Deriving the ALU from the Instruction Set & Shift (25 min.)
- Questions and Administrative Matters (3 min.)
- Multiply (20 min.)
- Questions and Break (5 min.)
- More Multiply (25 min.)

©DAP & SIK 1995

Deriving requirements of ALU

- Start with instruction set architecture: must be able to do all operations in ISA
- Tradeoffs of cost and speed based on frequency of occurrence, hardware budget
- MIPS ISA

©DAP & SIK 1995

MIPS arithmetic instructions

Instruction	*Example*	*Meaning*	*Comments*
add	**add $1,$2,$3**	**$1 = $2 + $3**	**3 operands; exception possible**
subtract	**sub $1,$2,$3**	**$1 = $2 – $3**	**3 operands; exception possible**
add immediate	**addi $1,$2,100**	**$1 = $2 + 100**	**+ constant; exception possible**
add unsigned	**addu $1,$2,$3**	**$1 = $2 + $3**	**3 operands; no exceptions**
subtract unsigned	**subu $1,$2,$3**	**$1 = $2 – $3**	**3 operands; no exceptions**
add imm. unsign.	**addiu $1,$2,100**	**$1 = $2 + 100**	**+ constant; no exceptions**
multiply	**mult $2,$3**	**Hi, Lo = $2 x $3**	**64-bit signed product**
multiply unsigned	**multu$2,$3**	**Hi, Lo = $2 x $3**	**64-bit unsigned product**
divide	**div $2,$3**	**Lo = $2 ÷ $3, Hi = $2 mod $3**	**Lo = quotient, Hi = remainder**
divide unsigned remainder	**divu $2,$3**	**Lo = $2 ÷ $3, Hi = $2 mod $3**	**Unsigned quotient &**
Move from Hi	**mfhi $1**	**$1 = Hi**	**Used to get copy of Hi**
Move from Lo	**mflo $1**	**$1 = Lo**	**Used to get copy of Lo**

©DAP & SIK 1995

MIPS logical instructions

Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

©DAP & SIK 1995

Compare and Branch

- Compare and Branch
  - BEQ rs, rt, offset if R[rs] == R[rt] then PC-relative branch
  - BNE rs, rt, offset <>
- Compare to zero and Branch
  - BLEZ rs, offset if R[rs] <= 0 then PC-relative branch
  - BGTZ rs, offset >
  - BLT <
  - BGEZ >=
  - BLTZAL rs, offset if R[rs] < 0 then branch and link (into R 31)
  - BGEZAL >=

©DAP & SIK 1995

MIPS ALU requirements

- Add, AddU, Sub, SubU, Addl, AddlU
  => 2's complement adder with overflow detection & inverter
- SLTI, SLTIU (set less than)
  => 2's complement adder with inverter, check sign bit of result
- BEQ, BNE (branch on equal or not equal)
  => 2's complement adder with inverter, check if result = 0
- And, Or, Andl, Orl
  => Logical AND, logical OR
- ALU from last lecture supports these ops

©DAP & SIK 1995

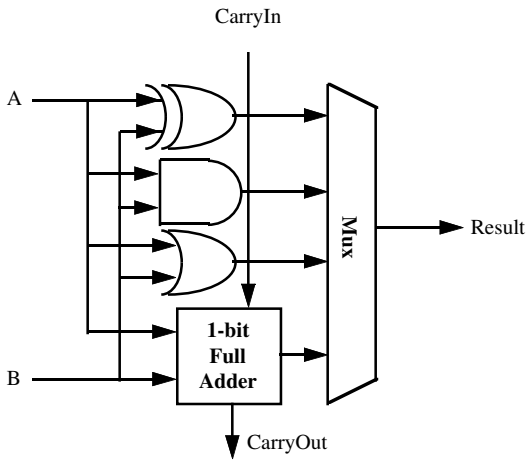
Additional MIPS ALU requirements

- **Xor, Nor, Xorl**
  => Logical XOR, logical NOR or use 2 steps: (A OR B) XOR 1111....1111
- **Sll, Srl, Sra**
  => Need left shift, right shift, right shift arithmetic by 0 to 31 bits
- **Mult, MultU, Div, DivU**
  => Need 32-bit multiply and divide, signed and unsigned

©DAP & SIK 1995

Add XOR to ALU

- **Expand Multiplexor**

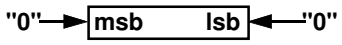


©DAP & SIK 1995

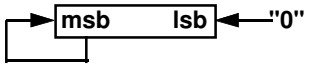
Shifters

Three different kinds:

***logical*-- value shifted in is always "0"**



***arithmetic*-- on right shifts, sign extend**



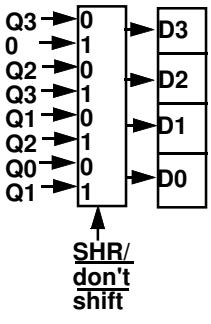
***rotating*-- shifted out bits are wrapped around (not in MIPS)**

Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

©DAP & SIK 1995

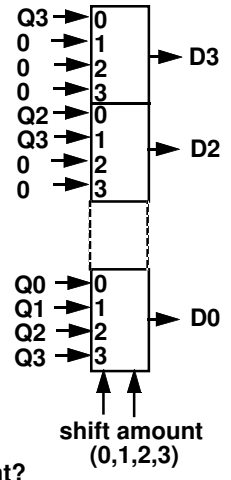
Multiplexor/Shifter

SHR:



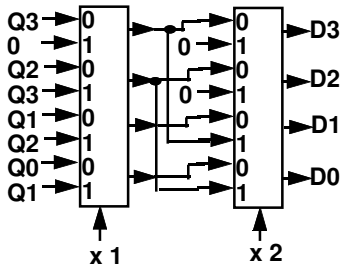
(5 inputs)

SHR 0, 1, 2, 3 bits:



4 x 4:1 Mux
1 stage

(7 inputs)

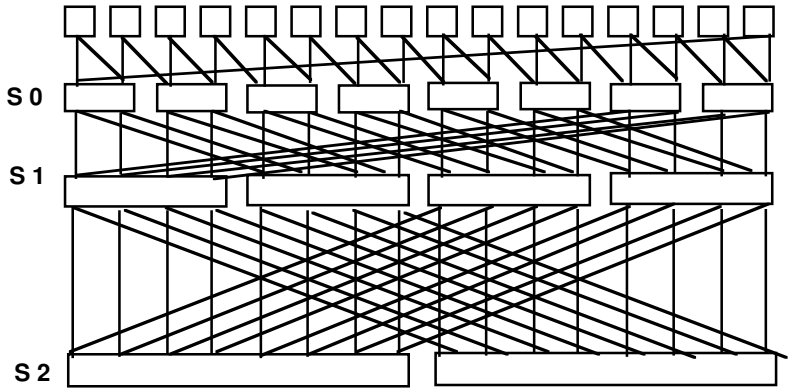


8 x 2:1 Mux
2 stages

How do arithmetic shift right?

©DAP & SIK 1995

General Scheme



Right-to-left connections support Rotate (not in MIPS but found in others)

©DAP & SIK 1995

32 Bit Shifter

Using this scheme for 32 bit data with 0-31 bit shifts would result in

5 stages of mux's (x1, x2, x4, x8, x16) if 2:1 are used

32 bits x 5 stages = 160 2:1 mux's!

3 stages of mux's (x4, x4, x2) if 2 levels of 4:1 and 1 level of 2:1 used

32 x 4:1 + 32 x 4:1 + 32 x 2:1

2 stages of mux's (x8, x4) if 1 level of 8:1 and 1 level of 4:1 used

32 x 8:1 + 32 x 4:1

©DAP & SIK 1995

Multibit Shifts (continued)

Mixed strategy, multiple control loops with more than one bit per loop

31 bit shift:
31 iterations with a 0,1 position shifter
11 iterations with a 0,1,2,3 position shifter
5 iterations with a 0,1,2,3,4,5,6,7 position shifter
3 iterations with an 0-15 position shifter

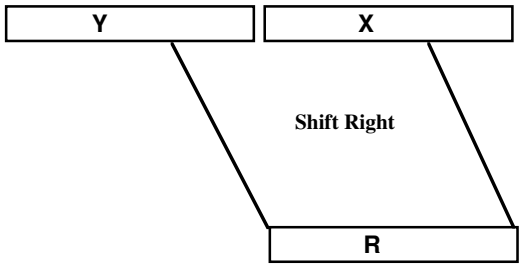
Fortunately, most shifts are relatively short (0-3 often implemented)

Extra Complexity: **can only do shift right so far**

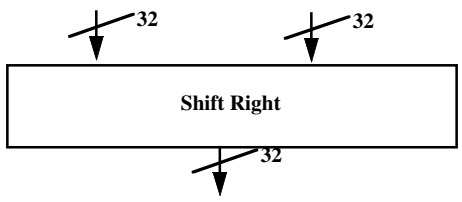
©DAP & SIK 1995

Funnel Shifter

Instead Extract 32 bits of 64.



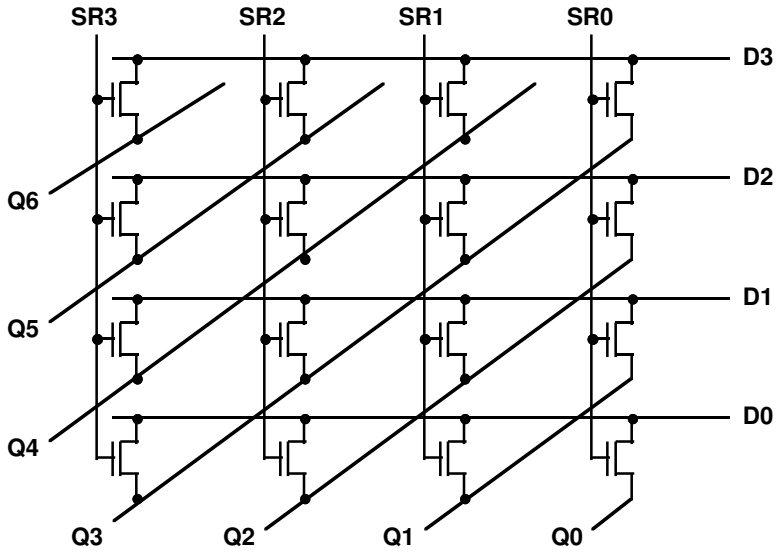
- **Logical: Y = 0**
- **Arithmetic?**
- **Rotate?**
- **Left shifts?**



©DAP & SIK 1995

Barrel Shifter

Technology-dependent solutions:



©DAP & SIK 1995

Administrative Matters

- Video tapes of lectures available for viewing in 205 McLaughlin, Mon. to Fri. 9 AM to 5 PM; Wed & Fri. 6 PM to 10 PM
- 1st Midterm 2 weeks, Feb. 22; 5 PM to 8 PM for 1.5 hour test in Sibley Auditorium;
  2 hand drawn pieces of paper
  (will be able to bring to second midterm too)
- LaVal's Afterwards to meet students, TAs, profs
- Other topics?

©DAP & SIK 1995

MULTIPLY

- Paper and pencil example:

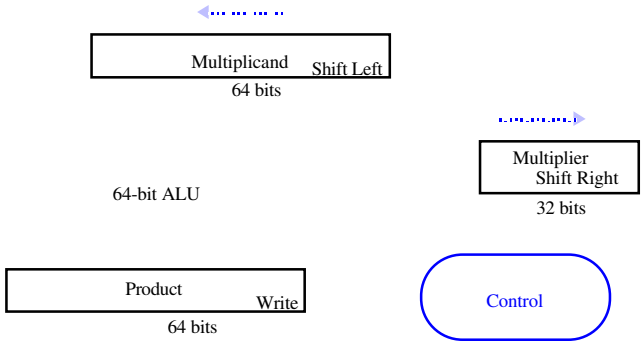
```
 Multiplicand          1000
Multiplier    x        1001
                       1000
                      0000
                     0000
                    1000
Product             1001000
```

- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place 0 (1 x multiplicand)
- 3 versions of multiply hardware & algorithm: successive refinement

©DAP & SIK 1995

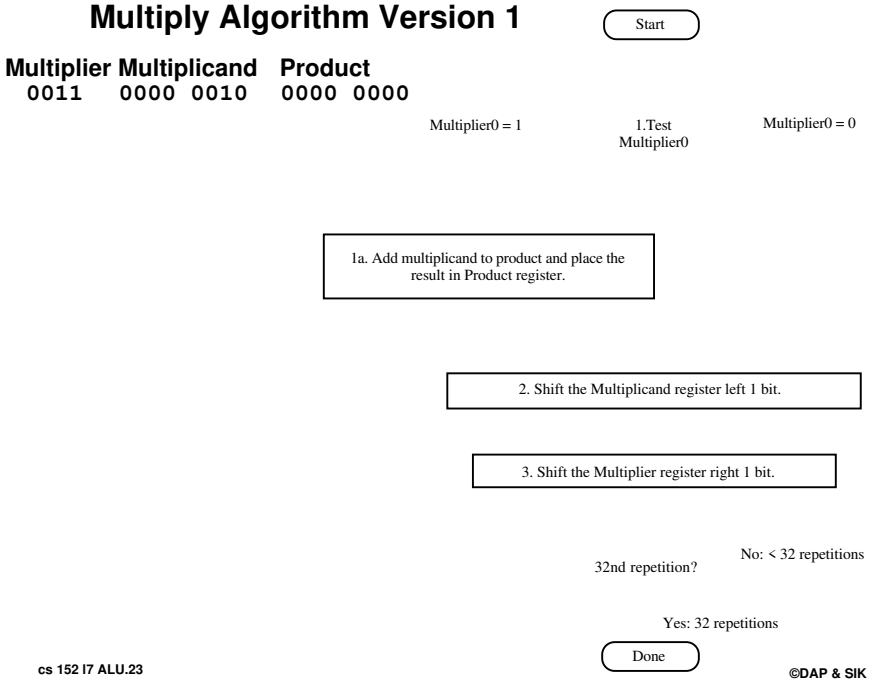
MULTIPLY HARDWARE Version 1

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



©DAP & SIK 1995

Multiply Algorithm Version 1



©DAP & SIK 1995

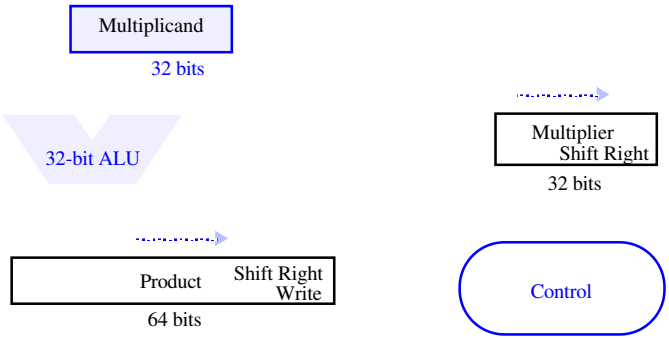
Observations on Multiply Version 1

- 1 clock per cycle => ≈ 100 clocks per multiply
  - Ratio of multiply to add 5:1 to 100:1
- 1/2 bits in multiplicand always 0
  => 64-bit adder is wasted
- 0's inserted in left of multiplicand as shifted
  => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?

©DAP & SIK 1995

MULTIPLY HARDWARE Version 2

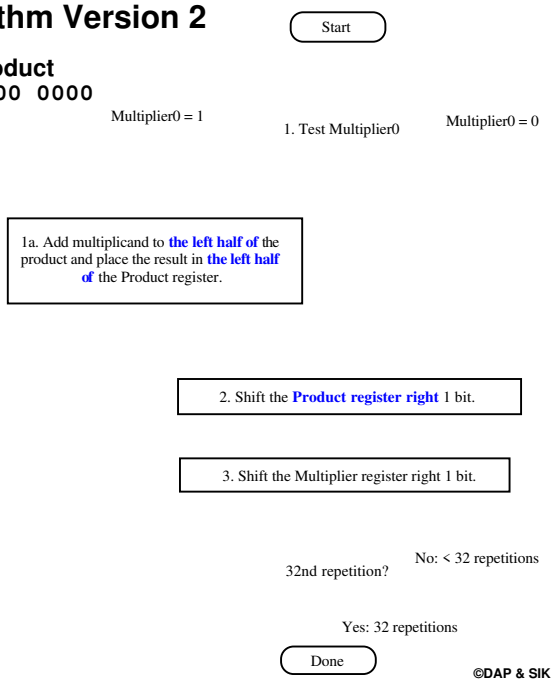
- **32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, 32-bit Multiplier reg**



©DAP & SIK 1995

Multiply Algorithm Version 2

Multiplier Multiplicand Product
0011 0010 0000 0000



©DAP & SIK 1995

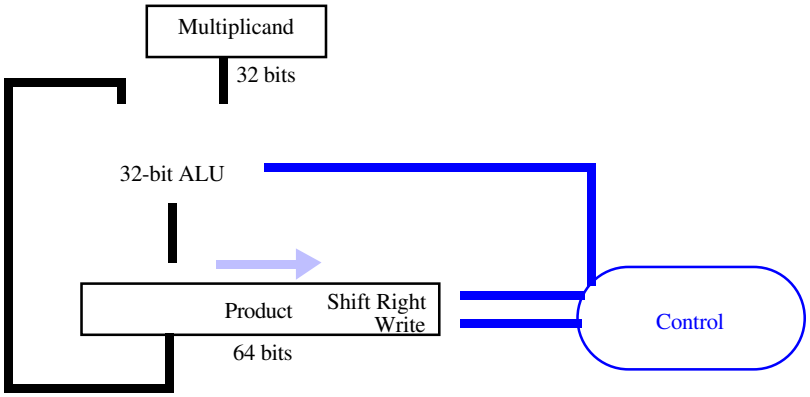
Observations on Multiply Version 2

- **Product register wastes space that exactly matches size of multiplier => combine Multiplier register and Product register**

©DAP & SIK 1995

MULTIPLY HARDWARE Version 3

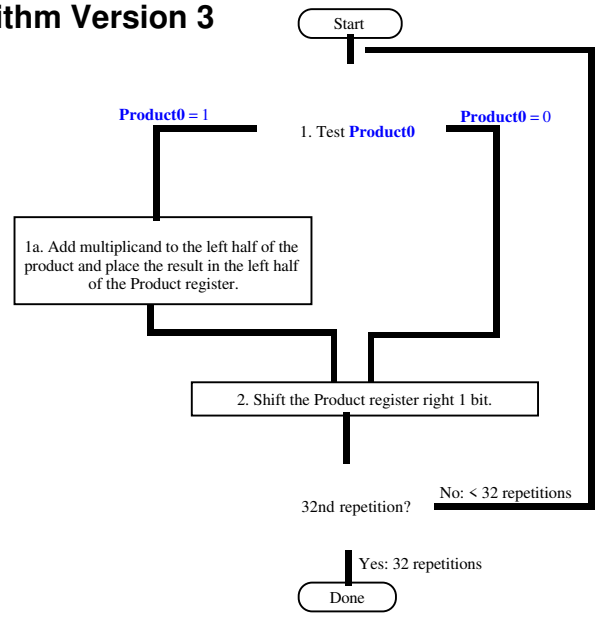
- **32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, (0-bit Multiplier reg)**



©DAP & SIK 1995

Multiply Algorithm Version 3

Multiplicand Product
`0010 0000 0011`



©DAP & SIK 1995

Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- What about signed multiplication?
  - easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)
  - Booth's Algorithm is more elegant way to multiply signed numbers using same hardware as before

©DAP & SIK 1995

Motivation for Booth's Algorithm

- Example 2 x 6 = 0010 x 0110:

```
          0010
x         0110
+         0000      shift (0 in multiplier)
+        0010       add (1 in multiplier)
+       0100        add (1 in multiplier)
+      0000         shift (0 in multiplier)
      00001100
```

- ALU with add or subtract gets same result in more than one way:

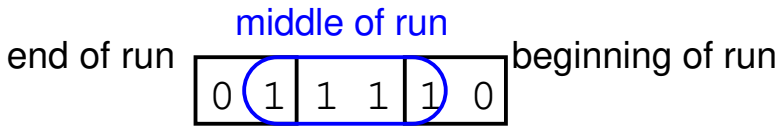
```
6          = - 2 + 8          , or
0110       = - 0010+ 1000
```

- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one. For example

```
          0010
x         0110
+         0000  shift (0 in multiplier)
-        0010   sub (first 1 in multiplier)
+       0000    shift (middle of string of 1s)
+      0010     add (prior step had last 1)
      00001100
```


©DAP & SIK 1995

Booth's Algorithm Insight



Current Bit	Bit to the Right	Explanation	Example
1	0	Beginning of a run of 1s	0001111000
1	1	Middle of a run of 1s	0001111000
0	1	End of a run of 1s	0001111000
0	0	Middle of a run of 0s	0001111000

Originally for Speed since shift faster than add for his machine

©DAP & SIK 1995

Booth's Algorithm

1. Depending on the current and previous bits, do one of the following:
   - 00: a. Middle of a string of 0s, so no arithmetic operations.
   - 01: b. End of a string of 1s, so add the multiplicand to the left half of the product.
   - 10: c. Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.
   - 11: d. Middle of a string of 1s, so no arithmetic operation.

2. As in the previous algorithm, shift the Product register right (arith) 1 bit.

Multiplicand	Product (2 x 3)	Multiplicand	Product (2 x -3)
0010	0000 0011 0	0010	0000 1101 0

©DAP & SIK 1995

Summary

- Instruction Set drives the ALU design
- Shifter: success refinement from 1/bit at a time shift register to barrel shifter
- Multiply: successive refinement to see final design
  - 32-bit Adder, 64-bit shift register, 32-bit Multiplicand Register
  - Booth's algorithm to handle signed multiplies
- There are algorithms that calculate many bits of multiply per cycle (see exercises 4.36 to 4.39 in COD)
- What's Missing from MIPS is Divide & Floating Point Arithmetic: Next time the Pentium Bug

©DAP & SIK 1995